

MPLAB® C32
C 编译器
用户指南



目录



MPLAB® C32 C 编译器 用户指南

前言

客户须知

所有文档均会过时，本文档也不例外。 **Microchip** 的工具和文档将不断演变以满足客户的需求，因此实际使用中有些对话框和 / 或工具说明可能与本文档所述之内容有所不同。请访问我们的网站（**www.microchip.com**）获取最新文档。

文档均标记有 “**DS**”编号。该编号出现在每页底部的页码之前。 **DS** 编号的命名约定为“**DSXXXXXA**”，其中 “**XXXXX**”为文档编号，“**A**”为文档版本。

欲了解开发工具的最新信息，请参考 **MPLAB® IDE** 在线帮助。从 **Help** （帮助）菜单选择 **Topics** （主题），打开现有在线帮助文件列表。

简介

本章包含在使用 MPLAB C32 C 编译器之前需要了解的一般信息。本章讨论的内容包括：

- 文档编排
- 本指南中使用的约定
- 推荐读物
- Microchip 网站
- 开发系统变更通知客户服务
- 客户支持
- 文档版本历史

文档编排

本文档介绍如何使用 MPLAB C32 C 编译器开发工具在目标电路板上仿真和调试固件。内容安排如下：

- **第 1 章 语言相关信息**——讨论 MPLAB C32 C 编译器命令行的使用、属性、pragma 伪指令和数据表示
- **第 2 章 库环境**——讨论 MPLAB C32 C 库的使用
- **第 3 章 中断**——对中断处理进行概述
- **第 4 章 低级处理器控制**——讨论 PIC32MX 器件的低级寄存器和配置的访问
- **第 5 章 编译器运行时环境**——讨论 MPLAB C32 C 编译器的运行时环境
- **附录 A 实现定义的操作**——讨论在 MPLAB C32 C 编译器中，对于“实现定义的操作”所作的选择
- **附录 B 开源许可**——简要介绍用于 MPLAB C32 C 编译器软件包某些部分的开源许可证

本指南中使用的约定

本手册使用如下文档约定：

文档约定

说明	涵义	示例
Arial 字体：		
斜体字	参考书目	*MPLAB® IDE User's Guide*
	需强调的文字	... *仅有*的编译器 ...
首字母大写	窗口	Output 窗口
	对话框	Settings 对话框
	菜单选项	选择 Enable Programmer
引号	窗口或对话框中的字段名	"Save project before build"
带右尖括号且有下划线的斜体文字	菜单路径	*File>Save*
粗体字	对话框按钮	单击 **OK**
	选项卡	单击 **Power** 选项卡
N'Rnnnn	verilog 格式的数，其中 N 是数字总数， R 是进制数， n 是一个数字。	4'b0010, 2'hF1
尖括号 < > 括起的文字	键盘上的按键	按 <Enter>， <F1>
Courier New 字体：		
常规 Courier New	源代码示例	`#define START`
	文件名	`autoexec.bat`
	文件路径	`c:\mcc18\h`
	关键字	`_asm, _endasm, static`
	命令行选项	`-Opa+, -Opa-`
	位值	`0, 1`
	常数	`0xFF, 'A'`
斜体 Courier New	可变参数	*`file`*`.o`，其中 *`file`* 可以是任一有效文件名
方括号 []	可选参数	`mcc18 [options]` *`file`* `[options]`
花括号和竖线：{ | }	选择互斥参数； "或"选择	`errorlevel {0|1}`
省略号 ...	代替重复文字	`var_name [, var_name...]`

	表示由用户提供的代码	`void main (void)`
		`{ ...`
		`}`

推荐读物

本用户指南介绍如何使用 MPLAB C32 C 编译器。下面列出了其他有用的文档。Microchip 提供了如下文档，推荐将这些文档作为补充参考资料。

Readme 文件

关于 Microchip 工具的最新信息，请阅读软件附带的相关 Readme 文件 （HTML 文件）。

针对器件的文档

Microchip 网站上提供了许多描述 16 位器件功能和特性的文档，其中包含：

- 具体器件以及器件系列的数据手册
- 器件系列的参考手册
- 程序员参考手册

MPLAB® C32 C Compiler Libraries （DS51685A）

MPLAB C32 库和预编译目标文件的参考指南。其中列出了随 MPLAB C32 C 编译器提供的所有库函数，以及它们详细的使用说明。

PIC32MX Configuration Settings （PIC32MX 配置设置）

其中列出了 MPLAB C32 C 编译器的 `#pragma config` 所支持的 Microchip PIC32MS 器件的配置位设置。

C 标准方面的信息

American National Standard for Information Systems – *Programming Language – C*. American National Standards Institute (ANSI), 11 West 42nd. Street, New York, New York, 10036.

此标准规定了用 C 语言编写程序的格式，并对 C 程序进行了解释。其目的是提高 C 程序在多种计算机系统上的可移植性、可靠性、可维护性及执行效率。

C 语言参考手册

Harbison, Samuel P. and Steele, Guy L., *C A Reference Manual*，第四版，Prentice-Hall, Englewood Cliffs, N.J. 07632.

Kernighan, Brian W. and Ritchie, Dennis M., *The C Programming Language*，第二版。 Prentice Hall, Englewood Cliffs, N.J. 07632.

Kochan, Steven G., *Programming In ANSI C*，修订版。 Hayden Books, Indianapolis, Indiana 46268.

Plauger, P.J., *The Standard C Library*, Prentice-Hall, Englewood Cliffs, N.J. 07632.

Van Sickle, Ted., *Programming Microcontrollers in C*，第一版。 LLH Technology Publishing, Eagle Rock, Virginia 24085.

GCC 文档

http://gcc.gnu.org/onlinedocs/

http://sourceware.org/binutils/

MICROCHIP 网站

Microchip 网站 （www.microchip.com）为客户提供在线支持。客户可通过该网站方便地获取文件和信息。只要使用常用的因特网浏览器即可访问。网站提供以下信息：

- **产品支持**——数据手册和勘误表、应用笔记和示例程序、设计资源、用户指南以及硬件支持文档、最新的软件版本以及存档软件
- **一般技术支持**——常见问题 （FAQ）、技术支持请求、在线讨论组以及 Microchip 顾问计划成员名单
- **Microchip 业务**——产品选型和订购指南、最新 Microchip 新闻稿、研讨会和活动安排表、 Microchip 销售办事处、代理商以及工厂代表列表

开发系统变更通知客户服务

Microchip 的客户通知服务有助于客户了解 Microchip 产品的最新信息。注册客户可在他们感兴趣的某个产品系列或开发工具发生变更、更新、发布新版本或勘误表时，收到电子邮件通知。

欲注册，请登录 Microchip 网站 www.microchip.com，点击 “变更通知客户（Customer Change Notification）” 服务并按照注册说明完成注册。

开发系统产品的分类如下：

- **编译器**——Microchip C 编译器及其他语言工具的最新信息，包括 MPLAB C18、MPLAB C30 和 MPLAB C32 C 编译器、 MPASM™ 和 MPLAB ASM30 汇编器、MPLINK™ 和 MPLAB LINK30 目标链接器、以及 MPLIB™ 和 MPLAB LIB30 目标库管理器。
- **仿真器**——Microchip在线仿真器的最新信息，包括MPLAB REAL ICE™和MPLAB ICE 2000 在线仿真器。
- **在线调试器**——Microchip在线调试器的最新信息，包括MPLAB ICD 2和PICkit™ 2。
- **MPLAB® IDE**——关于支持开发系统工具的 Windows® 集成开发环境 Microchip MPLAB IDE 的最新信息，主要针对 MPLAB IDE、 MPLAB IDE 项目管理器、MPLAB 编辑器、 MPLAB SIM 模拟器以及一般的编辑和调试功能。
- **编程器**——Microchip 编程器的最新信息，包括 MPLAB PM3 器件编程器以及 PICSTART® Plus、 PICkit™ 1 和 PICkit™ 2 开发编程器。

客户支持

Microchip 产品的用户可通过以下渠道获得帮助：

- 代理商或代表
- 当地销售办事处
- 应用工程师 （FAE）
- 技术支持

客户应联系其代理商、代表或应用工程师 （FAE）寻求支持。当地销售办事处也可为客户提供帮助。本文档后附有销售办事处的联系方式。

也可通过 http://support.microchip.com 获得网上技术支持。

文档版本历史

版本 A （2007 年 10 月）

- 本文档的第一版。

注：



第 1 章 语言相关信息

1.1 简介

本章讨论 MPLAB C32 C 编译器命令行的使用、属性、 pragma 伪指令和数据表示。

1.2 主要内容

本章讨论的内容包括：

- 概述
- 文件命名约定
- 数据存储
- 预定义宏
- 属性和 Pragma 伪指令
- 命令行选项
- 通过命令行编译单个文件
- 通过命令行编译多个文件

1.3 概述

编译驱动程序 （pic32-gcc）对 C 和汇编语言模块及库文件进行编译、汇编和链接。大多数编译器命令行选项对于 GCC 工具集的所有实现都是通用的。只有少数是专门针对 MPLAB C32 C 编译器的。

编译器命令行的基本形式如下：

```
pic32-gcc [options] files
```

注：	命令行选项和文件扩展名是区分大小写的。

在第 **1.8 节 “命令行选项”**中对可用的选项进行了描述。

例如，下面的命令行编译、汇编和链接 C 源文件 hello.c，生成可执行文件 hello.out。

```
pic32-gcc -o hello.out hello.c
```

1.4 文件命名约定

编译驱动程序识别如下文件扩展名，文件扩展名要区分大小写。

表 **1-1**： 文件名

扩展名	定义
file.c	必须预处理的 C 源文件。
file.h	头文件 （不对其进行编译或链接）。
file.i	已经过预处理的 C 源文件。
file.o	目标文件。
file.s	汇编语言源文件。

表 1-1： 文件名 （续）

扩展名	定义
file.s	必须预处理的汇编语言源文件。
其他	要传递给链接器的文件。

1.5 数据存储

1.5.1 存储尾数表示法

MPLAB C32 C 编译器以小尾数格式存储多字节值。即，最低有效字节存储在最低地址。

例如， 32 位值 0x12345678 将从地址 0x100 处开始，按如下形式存储：

地址	0x100	0x101	0x102	0x103
数据	0x78	0x56	0x34	0x12

1.5.2 整型表示

MPLAB C32 C 编译器中的整型值以 2 的补码形式表示，长度为 8 到 64 位。这些值可在通过 limits.h 编译后的代码中使用。

类型	位	最小值	最大值
char, signed char	8	-128	127
unsigned char	8	0	255
short, signed short	16	-32768	32767
unsigned short	16	0	65535
int, signed int, long, signed long	32	-2^{31}	$2^{31}-1$
unsigned int, unsigned long	32	0	$2^{32}-1$
long long, signed long long	64	-2^{63}	$2^{63}-1$
unsigned long long	64	0	$2^{64}-1$

1.5.3 有符号和无符号字符类型

默认情况下，不带修饰符的 char 类型的值是有符号值。根据 C 标准，这是由实现定义的操作，有一些环境[1] 将不带修饰符的 char 值定义为无符号值。对于给定的翻译单元，可以使用命令行选项 -funsigned-char 将默认类型设置为无符号。

1.5.4 浮点型表示

MPLAB C32 C 编译器使用 IEEE-754 浮点格式。 float.h 中提供了对于翻译单元的实现限制的详细信息。

类型	位
float	32
double	64
long double	64

1.5.5 指针

MPLAB C32 C 编译器中的指针长度均为 32 位。

1. 特别是 PowerPC 和 ARM

1.5.6 limits.h

limits.h 头文件定义了可以使用整型表示的值的范围。

宏名	值	说明
CHAR_BIT	8	最小非位域对象的大小 （以位数表示）。
SCHAR_MIN	-128	signed char 类型的对象可能具有的最小值。
SCHAR_MAX	127	signed char 类型的对象可能具有的最大值。
UCHAR_MAX	255	unsigned char 类型的对象可能具有的最大值。
CHAR_MIN	-128 （或 0，参见有符号和无符号字符类型）	char 类型的对象可能具有的最小值。
CHAR_MAX	127 （或 255，参见有符号和无符号字符类型）	char 类型的对象可能具有的最大值。
MB_LEN_MAX	16	任何语言环境中的多字节字符的最大长度。
SHRT_MIN	-32768	short int 类型的对象可能具有的最小值。
SHRT_MAX	32767	short int 类型的对象可能具有的最大值。
USHRT_MAX	65535	unsigned short int 类型的对象可能具有的最大值。
INT_MIN	-2^{31}	int 类型的对象可能具有的最小值。
INT_MAX	$2^{31}-1$	int 类型的对象可能具有的最大值。
UINT_MAX	$2^{32}-1$	unsigned int 类型的对象可能具有的最大值。
LONG_MIN	-2^{31}	long 类型的对象可能具有的最小值。
LONG_MAX	$2^{31}-1$	long 类型的对象可能具有的最大值。
ULONG_MAX	$2^{32}-1$	unsigned long 类型的对象可能具有的最大值。
LLONG_MIN	-2^{63}	long long 类型的对象可能具有的最小值。
LLONG_MAX	$2^{63}-1$	long long 类型的对象可能具有的最大值。
ULLONG_MAX	$2^{64}-1$	unsigned long long 类型的对象可能具有的最大值。

1.6 预定义宏

1.6.1 MPLAB C32 C 编译器宏

MPLAB C32 C 编译器定义了许多宏，大多数都带有前缀 “_MCHP_”，它们定义了各种目标特定选项、目标处理器和主机环境其他方面的特性。

_MCHP_SZINT	32 或 64，取决于设置整型大小的命令行选项（-mint32 -mint64）。
_MCHP_SZLONG	32 或 64，取决于设置整型大小的命令行选项（-mlong32 -mlong64）。
_MCHP_SZPTR	始终为 32，因为所有指针都是 32 位。
__mchp_no_float	如果指定了 -mno-float，则定义该宏。
__NO_FLOAT	如果指定了 -mno-float，则定义该宏。
__SOFT_FLOAT	如果未指定 -mno-float，则定义该宏。指示是否通过库调用浮点型。
__PIC__ __pic__	将翻译单元编译为位置无关的代码。
__PIC32MX __PIC32MX__	始终定义该宏。
PIC32MX	如果未指定 -ansi，则定义该宏。
__LANGUAGE_ASSEMBLY __LANGUAGE_ASSEMBLY__ _LANGUAGE_ASSEMBLY	如果是编译经过预处理的汇编文件（.s 文件），则定义该宏。
LANGUAGE_ASSEMBLY	如果是编译经过预处理的汇编文件（.s 文件），并且未指定 -ansi，则定义该宏。
__LANGUAGE_C __LANGUAGE_C__ _LANGUAGE_C	如果是编译 C 文件，则定义该宏。
LANGUAGE_C	如果是编译 C 文件，并且未指定 -ansi，则定义该宏。
__*processor*__	其中，“processor”是传递给 -mprocessor 选项的参数的大写形式。例如，-mprocessor=32mx12f3456 将定义 __32MX12F3456__。

1.6.2 SDE 兼容性宏

MIPS® SDE（软件开发环境）定义了许多宏，大多数带有前缀 “_MIPS_”，它们定义了各种目标特定选项的特性，其中一些由命令行选项决定（例如，-mint64）。在适用时，这些宏将由 MPLAB C32 C 编译器定义，以方便将应用程序和中间件从 SDE 移植到 MPLAB C32 C 编译器。

_MIPS_SZINT	32 或 64，取决于设置整型大小的命令行选项（-mint32 -mint64）。
_MIPS_SZLONG	32 或 64，取决于设置整型大小的命令行选项（-mlong32 -mlong64）。
_MIPS_SZPTR	始终为 32，因为所有指针都是 32 位。
__mips_no_float	如果指定了 -mno-float，则定义该宏。

__mips__ _mips _MIPS_ARCH_PIC32MX _MIPS_TUNE_PIC32MX _R3000 __R3000 __R3000__ __mips_soft_float __MIPSEL __MIPSEL__ _MIPSEL	始终定义该宏。
R3000 MIPSEL	如果未指定 -ansi，则定义该宏。
_mips_fpr	定义为 32。
__mips16 __mips16e	如果指定了 -mips16 或 -mips16e，则定义该宏。
__mips	定义为 32。
__mips_isa_rev	定义为 2。
_MIPS_ISA	定义为 _MIPS_ISA_MIPS32。
__mips_single_float	如果指定了 -msingle-float，则定义该宏。

1.7 属性和 PRAGMA 伪指令

1.7.1 函数属性

always_inline

如果函数声明为 inline，则始终内联函数，即使是未指定任何优化级别。

longcall

始终通过以下方式调用函数：首先将其地址装入一个寄存器，然后使用该寄存器的内容进行调用。这使得可以调用位于直接调用指令 28 位寻址范围之外的函数。

far

在功能上等价于 longcall。

near

始终使用绝对调用指令来调用函数，即使是指定了 -mlong-calls 命令行选项。

mips16

以 MIPS16 指令集为函数生成代码。

nomips16

始终以 MIPS32 指令集为函数生成代码，即使是编译带有 -mips16 命令行选项的翻译单元。

interrupt

为用作中断处理程序的函数生成序言 （prologue）和尾声 （epilogue）代码。请参见**第 3 章 “中断”**和**第 3.5 节 “异常处理程序”**。

vector

在所指示的异常向量 （其目标为一个函数）处生成一条转移指令。请参见**第 3 章 “中断”**和**第 3.5 节 “异常处理程序”**。

at_vector

将函数体放到所指示的异常向量地址处。请参见**第 3 章 “中断”**和**第 3.5 节 “异常处理程序”**。

naked

不为函数生成序言或尾声代码。

section (“name”)

将函数放入由 “name” 指定的段。

例如，

```
void __attribute__ ((section (".wilma"))) baz () {return;}
```

函数 baz 将被放入 .wilma 段。

-ffunction-sections 命令行选项对使用 section 属性定义的函数不起作用。

unique_section

将函数放入唯一指定的段中，就如同指定了 -ffunction-sections。如果函数还具有 section 属性，那么将使用相应的段名作为前缀来生成唯一的段名。

例如，

```
void __attribute__ ((section (".fred"), unique_section) foo (void) {return;}
```

函数 foo 将被放入 .fred.foo 段。

noreturn

向编译器指示，函数将永不返回。在一些情况下，这将使编译器可以在调用函数中生成效率更高的代码，因为在执行优化时可以无需考虑函数确实会返回时的行为。声明为 noreturn 的函数的返回类型应始终为 void。

noinline

始终不考虑将函数内联。

pure

如果某个函数除了对其返回值之外没有任何其他副面影响，并且返回值仅依赖于参数和 / 或（非易变）全局变量，那么对于该函数的调用，编译器可以执行更积极的优化。此类函数可以使用 pure 属性指示。

const

如果一个 pure 函数仅根据其参数决定其返回值（即，不检查任何全局变量），那么可以将其声明为 const，以允许更积极的优化。请注意，对指针参数进行解引用的函数不属于 const，因为指针解引用使用了不属于参数的值，虽然指针本身是一个参数。

format (type, format_index, first_to_check)

format 属性指示函数采用 printf、scanf、strftime 或 strfmon 样式的格式字符串和参数，编译器应当根据格式字符串对那些参数进行类型检查，方法与针对标准库函数时一样。

type 参数是 printf、scanf、strftime 或 strfmon 的其中之一（可以选择在两端附加双下划线，例如， __printf__），决定格式字符串的解释方式。

format_index 参数指定哪个函数参数是格式字符串。函数参数从最左端的参数开始，从 1 开始编号。

first_to_check 参数指定根据格式字符串检查的第一个参数的编号。如果 first_to_check 为 0，那么不执行类型检查，编译器仅检查格式字符串的一致性（例如， vfprintf）。

format_arg (index)

format_arg 属性指定函数处理 printf 样式的格式字符串，编译器应检查格式字符串的一致性。用作格式字符串的函数属性使用 index 标识。

nonnull (index, ...)

向编译器指示，传递给函数的一个或多个指针参数必须为非空指针。如果编译器确定有空指针作为值传递给非空参数，并且指定了 -Wnonnull 命令行选项，那么它会发出警告诊断信息。

如果未为 nonnull 属性提供任何参数，那么函数的所有指针参数均标记为非空。

unused

向编译器指示，函数可能不使用。如果该函数未被使用，编译器将不会发出警告。

used

向编译器指示，将始终使用该函数，即使编译器无法检查到对该函数的引用，也必须为函数生成代码。例如，行内汇编代码是到某个静态函数的唯一引用。

deprecated

在使用指定为 deprecated 的函数时，会产生警告。

warn_unused_result

如果调用程序未使用所指示函数的返回值，将会发出警告。

weak

weak 符号指示，如果有另一个版本的相同符号可用，那么应使用另一个版本。例如，要使已实现的库函数可以由用户编写的函数覆盖，可以使用该属性。

malloc

所指示函数的任何非空指针返回值将不会赋给函数返回时有效的任何其他指针。这使编译器可以改善优化。

alias (“symbol”)

指示函数是另一个符号的别名。例如，

```
void foo (void) { /* stuff */ }
void bar (void) __attribute__ ((alias(“foo”)));
```

符号 bar 被视为是符号 foo 的别名。

1.7.2 变量属性

aligned (n)

具有该属性的变量将在下 n 字节边界处对齐。

aligned 属性还可以用于结构成员。此类成员将在结构内对齐到所指示的边界处。

如果省略了对齐值 n，那么变量的对齐值设置为 8 （基本数据类型的最大对齐值）。

请注意， aligned 属性是用于增大变量的对齐值，而不是减小它。要减小变量的对齐值，请使用 packed 属性。

cleanup (function)

指示当具有该属性的自动函数作用域变量超出作用域时要调用的函数。

所指示的函数应只具有一个参数，即指向与具有该属性的变量的类型兼容的指针，返回类型应为 void。

deprecated

在使用指定为 deprecated 的变量时，将会产生警告。

packed

具有该属性的变量或结构成员将具有所可能的最小对齐值。即，将不为声明分配任何对齐填充存储空间。与 aligned 属性联合使用时， packed 可以用于设置任意的对齐限制，即大于或小于变量或结构成员的类型所具有的默认对齐值。

section (“name”)

将函数放入由 “name”指定的段。

例如，

```
unsigned int dan __attribute__ ((section (“.quixote”)))
```

变量 dan 将被放入 .quixote 段。

除非同时也指定了 unique_section，否则 -fdata-sections 命令行选项对使用 section 属性定义的变量不起作用。

unique_section

将变量放入唯一指定的段中，就如同指定了 -fdata-sections。如果变量还具有 section 属性，那么将使用相应的段名作为前缀来生成唯一的段名。

例如，

```
int tin __attribute__ ((section (“.ofcatfood”), unique_section)
```

变量 tin 将被放入 .ofcatfood 段。

transparent_union

如果联合类型的函数形参带有 transparent_union 属性，那么传递相应的实参时，它的类型视同为联合的第一个成员的类型。

unused

向编译器指示，变量可能不使用。如果该变量未被使用，编译器将不会发出警告。

weak

weak 符号指示，如果有另一个版本的相同符号可用，那么应使用另一个版本。

1.7.3 Pragma 伪指令

```
#pragma interrupt
```

将一个函数标记为中断处理程序。函数的序言或尾声代码将执行范围更广的现场保护。请参见**第 3 章 “中断”**和**第 3.5 节 “异常处理程序”**。

```
#pragma vector
```

在所指示的异常向量（其目标为一个函数）处生成一条转移指令。请参见**第 3 章 “中断”**和**第 3.5 节 “异常处理程序”**。

`#pragma config`

`#pragma config` 伪指令指定要由应用程序使用的特定于处理器的配置设置（即，配置位）。请参见**第 4 章 “低级处理器控制”**。

1.8 命令行选项

MPLAB C32 C 编译器提供了许多控制编译的选项，它们都是区分大小写的。

- 针对 PIC32MX 器件的选项
- 控制输出类型的选项
- 控制 C 语言的选项
- 控制警告与错误的选项
- 调试选项
- 控制优化的选项
- 控制预处理器的选项
- 汇编选项
- 链接选项
- 目录搜索选项
- 代码生成约定选项

1.8.1 针对 PIC32MX 器件的选项

表 1-2： 针对 PIC32MX 器件的选项

选项	定义
`-m`*`processor`*	选择编译所针对的器件 （例如， `-m`*`processor`*=32MX360F512L）
`-mips16` `-mno-mips16`	生成（不生成） MIPS16 代码。
`-mno-float`	不使用浮点库。
`-msingle-float`	假定浮点协处理器仅支持单精度运算。
`-mdouble-float`	假定浮点协处理器支持双精度运算。这是默认设置。
`-mlong64`	强制 `long` 类型为 64 位宽。关于默认值和确定指针长度的方式的说明，请参见 `-mlong32`。
`-mlong32`	强制 `long`、 `int` 和 `pointer` 类型为 32 位宽。 `int`、 `long` 和 `pointer` 的默认长度为 32 位。
`-G` *`num`*	将长度小于等于 *`num`* 字节的全局和静态项放入小模式数据或 bss 段，而不是放入一般数据或 bss 段。这使得可以使用单条指令对数据进行访问。 所有模块应使用相同的 `-G` *`num`* 值编译。

表 **1-2：** 针对 **PIC32MX** 器件的选项 （续）

选项	定义
`-membedded-data` `-mno-embedded-data`	在可能的情况下，首先将变量分配到只读数据段，然后在可能的情况下，接着将变量分配到小模式数据段，否则分配到数据段中。这种情况下生成的代码会比默认情况下生成的代码略慢，但可以减少在执行时所需的 RAM 量，所以对于一些嵌入式系统可能是更好的选择。
`-muninit-const-in-rodata` `-mno-uninit-const-in-rodata`	将非初始化 `const` 变量放入只读数据段。该选项只有在与 `-membedded-data` 联合使用时才有意义。
`-mcheck-zero-division` `-mno-check-zero-division`	整数被 0 除时产生 （不产生）陷阱。默认设置为 `-mcheck-zero-division`。
`-mmemcpy` `-mno-memcpy`	对于非平凡块移动强制 （不强制）使用 `memcpy()`。默认设置是 `-mno-memcpy`，这允许 GCC 内联大多数固定大小的拷贝。
`-mlong-calls` `-mno-long-calls`	禁止 （不禁止）使用 `jal` 指令。使用 `jal` 调用函数效率更高，但要求调用程序和被调用程序处于同一个 256 MB 的段中。 该选项对于 abicalls 代码不起作用。默认设置为 `-mno-long-calls`。
`-mno-peripheral-libs`	链接时不使用标准外设库。

1.8.2 控制输出类型的选项

下面的选项控制编译器的输出类型。

表 1-3： 输出类型控制选项

选项	定义
-c	编译或汇编源文件，但不链接。默认的文件扩展名为 .o。
-E	在预处理过程之后，即正常运行编译器之前停止。默认输出文件为 stdout。
-o *file*	将输出放在 *file* 中。
-S	在正常编译之后，即调用汇编器之前停止。默认输出文件扩展名为 .s。
-v	在编译的每个阶段打印执行的命令。
-x	可用 -x 选项显式地指定输入语言： -x *language* 为后面的输入文件显式地指定语言 （而不是让编译器根据文件后缀名选择默认的语言）。该选项适用于其后直到下一个 -x 选项之前的所有输入文件。 MPLAB C32 C 编译器支持下面的值： c c-header cpp-output assembler assembler-with-cpp -x none 关闭所有语言指定，随后的文件将按其后缀名处理。 -x none 选项是默认的，但如果已使用另一个 -x 选项，则还必须明确使用 -x none。例如： pic32-gcc -x assembler foo.asm bar.asm -x none main.c mabonga.s 没有 -x none 时，编译器将假定所有输入文件都为汇编语言编写的程序。
--help	打印命令行选项的描述。

1.8.3 控制 C 语言的选项

下面的选项定义编译器使用的 C 语言类型。

表 1-4： C 语言控制选项

选项	定义
-ansi	支持 （且仅支持）所有 ANSI 标准的 C 程序。
-aux-info filename	对于所有在翻译单元中声明和 / 或定义的函数，包括头文件中的函数，输出到给定文件名的原型声明中。除了 C，该选项在其他语言中通常被忽略。除了声明以外，文件在注释中指出了每个声明的来源 （源文件和行），不论声明是隐含的、原型的，还是非原型的 （在行号和冒号后面的第一个字符中， I、 N 代表新的， O 代表旧的），也不论它来自声明还是定义 （在随后的字符中，分别用 C 和 F 代表）。如果是函数定义，在函数声明之后的注释中，还提供 K&R 型参数列表，后跟这些参数的声明。
-ffreestanding	指明编译在独立环境中进行。这暗指 -fno-builtin 选项。独立的环境就是其中可能不存在标准库，程序也不必在主函数中启动的环境。最显而易见的例子就是 OS 内核。这与 -fno-hosted 等价。
-fno-asm	不识别 asm、 inline 或 typeof 关键字，因此代码可以将这些单词用作标识符。可以使用关键字 __asm__、__inline__ 和 __typeof__。 -ansi 暗指 -fno-asm。
-fno-builtin -fno-builtin-*function*	不识别不以 __builtin_ 作为前缀开始的内建函数。
-fsigned-char	使 char 型变量为有符号，就像 signed char。 **（这是默认设置。）**
-fsigned-bitfields -funsigned-bitfields -fno-signed-bitfields -fno-unsigned-bitfields	如果声明时未使用 signed 或 unsigned，这些选项用来控制位域是有符号还是无符号的。默认情况下，这样的位域都是有符号的，除非使用 -traditional，它使位域总是无符号的。
-funsigned-char	使 char 型变量为无符号，就像 unsigned char。
-fwritable-strings	将字符串存储到可写的数据段中，但不要使字符串唯一。

1.8.4 控制警告与错误的选项

警告是诊断消息，它报告非本质错误、但有危险的语法结构，或暗示可能存在错误。

可以使用以 -W 开头的选项请求许多特定的警告，例如，使用 -Wimplicit 请求关于隐式声明的警告。这些特定的警告选项全部可以用以 -Wno- 开头的否定形式来关闭警告，如 -Wno-implicit。本手册只列出了这两种形式中的一种，这两种形式都不是默认的。

下面的选项控制 MPLAB C32 C 编译器产生的警告的数量和种类。

表 1-5： **-WALL** 隐含的警告与错误选项

选项	定义
-fsyntax-only	检查代码的语法，除此之外不做任何事情。
-pedantic	发出严格 ANSI C 要求的所有警告。拒绝所有使用禁止扩展名的程序。
-pedantic-errors	类似于 -pedantic，只是发出错误而不是警告。
-w	禁止所有警告消息。
-Wall	使能本表中列出的所有 -W 选项。这将使能关于某些用户认为有问题的，及容易避免的 （或修改来禁止警告的）语法结构的所有警告，即使是与宏一起。
-Wchar-subscripts	如果数组下标具有 char 类型则警告。
-Wcomment -Wcomments	当注释开始符号 /* 出现在 /* 注释中，或反斜杠 - 换行出现在 // 注释中发出警告。
-Wdiv-by-zero	编译时发现整数除以 0 则警告。要禁止这个警告消息，可以使用 -Wno-div-by-zero。浮点数除以 0 不会警告，因为它可以是获得无穷大和 NaN 的一种合法方法。 **（这是默认设置。）**
-Werror-implicit-function-declaration	函数在声明前被使用将给出错误。
-Wformat	检查对 printf 和 scanf 等函数的调用，确保所提供参数的类型与指定的格式字符串相符合。
-Wimplicit	等价于同时指定 -Wimplicit-int 和 -Wimplicit-function-declaration。
-Wimplicit-function-declaration	函数在声明前被使用将给出警告。
-Wimplicit-int	如果声明没有指定类型则警告。
-Wmain	如果 main 的类型有问题则警告。 main 应该是一个具有外部链接的函数，它返回 int，并带有正确类型的 0、 2 或 3 个参数。

-Wmissing-braces	如果一个聚集或联合的初始化中括号不全则警告。在下面的例子中， a 的初始化中括号不全，而对 b 的初始化是正确的。
	`int a[2][2] = { 0, 1, 2, 3 };`
	`int b[2][2] = { { 0, 1 }, { 2, 3 } };`

表 1-5： **-WALL** 隐含的警告与错误选项 （续）

选项	定义

-Wmulticharr
-Wno-multichar
	使用多字符的 *character* 常量时警告。通常出现这样的常量是由于输入错误。由于这种常量具有实现定义的值，不应将它们用在可移植代码中。下面举例说明了多字符 character 常量的使用：
	char
	xx(void)
	{
	return('xx');
	}

-Wparentheses	在某些上下文中省略圆括号时警告，如在需要真值的上下文中有一个赋值，或者在运算符嵌套的运算中，人们往往辨别不清运算的优先级。
-Wreturn-type	当函数定义为其返回值类型默认为 int 时发出警告。如果函数的返回值类型不是 void，那么不带返回值的任何 return 语句都会导致产生警告。
-Wsequence-point	由于违背 C 标准中的顺序点规则而导致代码中有未定义的语义时发出警告。 C 标准定义了 C 程序中根据顺序点对表达式求值的顺序，顺序点代表程序各部分执行的局部顺序：在顺序点之前执行的部分和顺序点之后执行的部分。这些在一个完整表达式 （不是一个更大的表达式的一部分）的求值之后，在对第一个运算符 （&&、 ||、 ?: 或 , （逗号）运算符）求值之后，在调用函数前 （但在对其参数和表示被调用函数的表达式求值后），以及某些其他地方发生。除了顺序点规则指定的顺序外，未指定表达式的子表达式的求值顺序。所有这些规则仅规定了局部的顺序，而没有规定全局的顺序，因为，如在一个表达式中调用了两个函数，而它们之间没有顺序点，就没有指定函数调用的顺序。但是，标准委员会规定函数调用不能重叠。 没有指定在顺序点之间，对对象的值的修改何时生效。操作依赖于这一点的程序有不确定的操作； C 标准规定，“在上一个顺序点和下一个顺序点之间，对象所储存的值最多被表达式求值修改一次。而且，前一个值是只读的以便确定将被储存的值。”如果程序违反这些规则，任何特定实现的结果都是完全不可预估的。 具有未定义操作的代码示例有 a = a++;、a[n] = b[n++] 及 a[i++] = i;。该选项不能诊断某些更复杂的情况，并可能给出偶然错误的结果，但通常在检测程序中的这类问题时，该选项还是很有效的。

表 **1-5：** **-WALL** 隐含的警告与错误选项 （续）

选项	定义
-Wswitch	每当 switch 语句中有一个枚举类型的索引，并且这个枚举的一个或多个指定码缺少 case 时发出警告。（默认标号的存在禁止该警告。）当使用该选项时，枚举范围之外的 case 标号也会引起警告。
-Wsystem-headers	打印关于系统头文件中语法结构的警告消息。系统头文件的警告通常是被禁止的，因为通常认为它们不会有真正的问题，只会使编译器的输出可读性更差。使用这个命令行选项告知 MPLAB C32 C 编译器发出关于系统头文件的警告，就像在用户代码中一样。但是，注意将 -Wall 与该选项一起使用时不会对系统头文件中的未知 pragma 伪指令发出警告。这时，必须同时使用 -Wunknown-pragmas。
-Wtrigraphs	遇到三字母组合时发出警告 （假定使能了三字母组合）。
-Wuninitialized	使用自动变量而没有先对其初始化时发出警告。 这些警告只有在允许优化时才出现，因为它们需要只有优化时才计算的数据流信息。 仅当将变量分配给寄存器时才产生这些警告。因此，对于声明为 volatile 的变量，或是变量地址被占用，或者大小不是 1、 2、 4 或 8 字节的变量不会产生这些警告。同样对于结构、联合或数组，即使它们在寄存器中，也不会产生这些警告。 注意，当一个变量只是用于计算一个值而变量本身不会被使用时，也不会产生警告，因为在警告被打印前，这样的计算就会被数据流分析删除。
-Wunknown-pragmas	当遇到一个 MPLAB C32 C 编译器无法理解的 #pragma 伪指令时发出警告。如果使用该命令行选项，甚至对系统头文件中的未知 pragma 伪指令也会发出警告。如果警告只能通过 -Wall 命令行选项来使能，情况就不是这样了。
-Wunused	每当变量除了其声明外未被使用过时，每当函数声明为 static 但从未定义时，每当声明了标号但未使用时，每当一条语句的计算结果未被显式使用时，发出警告。 要获得未使用的函数参数的警告，必须同时指定 -W 和 -Wunused。 强制转换表达式类型可以避免禁止对表达式的这种警告。同样地， unused 属性可以禁止对未使用的变量、参数和标号的警告。
-Wunused-function	每当声明了 static 函数但没有定义函数时，或一个非内联 static 函数未使用时，发出警告。
-Wunused-label	声明了一个标号但未使用时发出警告。要禁止这种警告，可以使用 unused 属性。

表 1-5： **-WALL** 隐含的警告与错误选项 （续）

选项	定义
-Wunused-parameter	当对函数参数进行了声明但从未使用时，发出警告。要禁止这种警告，可以使用 unused 属性。
-Wunused-variable	当对局部变量或非常量的 static 变量进行了声明但从未使用时，发出警告。要禁止这种警告，可以使用 unused 属性。
-Wunused-value	语句的计算结果未显式使用时发出警告。要禁止这种警告，可以将表达式类型强制转换为 void。

下面是不被 -Wall 隐含的 -W 选项。其中有些是关于用户通常认为不会有问题，但有时希望检查一下的语法结构的警告。其他是在某些情况下必须或很难避免的语法结构的警告，没有简单的方法来修改代码以禁止这些警告。

表 1-6： -WALL 不隐含的警告与错误选项

选项	定义

-W	为以下事件输出额外警告消息：
	• 非易变的自动变量可能会被对 longjmp 的调用改变。这些警告仅在优化编译时才会出现。编译器仅识别对 setjmp 的调用，而不会知道将在何处调用 longjmp，信号处理程序可以在代码中的任何地方调用 longjmp。因此，即使当实际没有问题时也可能会产生警告，因为实际上不能在会产生问题的地方调用 longjmp。
	• 函数可以通过 return value; 和 return; 退出。函数体结束时不传递任何返回值的语句视为 return;。
	• 表达式语句或者逗号表达式的左侧没有副作用。要禁止这种警告，可以将未使用表达式的类型强制转换为 void。例如，表达式 x[i,j] 会产生警告，而表达式 x[(void)i,j] 不会产生警告。
	• 用 < 或 <= 将无符号值与 0 比较。
	• 出现了像 x<=y<=z 这样的不等式，这等价于 (x<=y ? 1 : 0) <= z，这只是普通数学表示的不同解释罢了。
	• 存储类型修饰符如 static 在声明中没有放在最前面，根据 C 标准，这种用法已经过时了。
	• 如果还指定了 -Wall 或 -Wunused，会出现关于未使用变量的警告。
	• 当将有符号值转换为无符号值时，比较有符号值和无符号值会产生不正确的结果。（但是如果还指定了 -Wno-sign-compare 的话，就不会产生警告。）
	• 聚集的初始化中括号不全。例如，下面的代码由于在初始化 x.h 时漏掉了括号会产生警告：
	`struct s { int f, g; };`
	`struct t { struct s h; int i; };`
	`struct t x = { 1, 2, 3 };`
	• 聚集的初始化中没有初始化所有成员。例如，下面的代码由于 x.h 会被隐式初始化为零而会产生警告：
	`struct s { int f, g, h; };`
	`struct s x = { 3, 4 };`

-Waggregate-return	定义或调用了返回结构或联合的任何函数时产生警告。
-Wbad-function-cast	当将函数调用强制转换为不匹配类型时产生警告。例如，如果 int foof() 被强制转换为任何 * 指针类型会产生警告。

表 1-6： -WALL 不隐含的警告与错误选项 （续）

选项	定义
-Wcast-align	当强制转换指针类型，使目标所需分配的存储空间增加时产生警告。例如，如果 char * 被强制转换为 int * 会产生警告。
-Wcast-qual	当强制转换指针类型，从目标类型中去除类型限定符时产生警告。例如，如果 const char * 被强制转换为不带修饰符的的 char * 会产生警告。
-Wconversion	如果一个原型导致一个参数的类型转换与没有原型时不同，则发出警告。这包括定点型转换为浮点型或反之，及改变定点参数符号或宽度的转换，与默认的提升相同时除外。 当负的整型常量表达式隐式转换为无符号类型时也发出警告。例如，如果 x 为无符号类型，赋值 x = -1 将产生警告。但是，显式的强制类型转换，如 (unsigned) -1，不会产生警告。
-Werror	使所有警告变为错误。
-Winline	一个函数已声明为内联，或指定了 -finline-functions 选项时，如果函数不能被内联，将产生警告。
-Wlarger-than-*len*	当定义了大于 *len* 字节的对象时产生警告。
-Wlong-long -Wno-long-long	使用 long long 类型时发出警告。这是默认设置。要禁止这个警告消息，可以使用 -Wno-long-long。仅当使用 -pedantic 标志时，才考虑标志 -Wlong-long 和 -Wno-long-long。
-Wmissing-declarations	如果全局函数在定义之前没有先对其进行声明会产生警告。即使定义本身提供了原型，也要在定义全局函数之前先声明它。
-Wmissing-format-attribute	如果使能了 -Wformat，可指定 format 属性的函数也会产生警告。注意这些函数仅是可指定这一属性的函数，并不是已指定这一属性的函数。如果不使能 -Wformat，该选项不起作用。
-Wmissing-noreturn	对可指定 noreturn 属性的函数产生警告。这些函数仅是可指定这一属性的函数，并不是已指定这一属性的函数。手工检验这些函数时要小心。实际上，在添加 noreturn 属性之前也不要返回；否则可能会引入微小的代码生成错误。
-Wmissing-prototypes	如果全局函数在定义之前没有先声明原型会产生警告。即使定义本身提供了原型也会发出这个警告。（该选项可用于检测不在头文件中声明的全局函数。）
-Wnested-externs	如果在函数内部遇到了 extern 声明，发出警告。
-Wno-deprecated-declarations	不要对使用通过 deprecated 属性指定为 deprecated 的函数、变量和类型发出警告。
-Wpadded	如果一个结构中包含了填充，不管是为了对齐结构的一个元素，还是为了对齐整个结构，都发出警告。
-Wpointer-arith	对于与函数类型或 void 的长度有关的任何类型发出警告。为方便使用 void * 指针和指向函数的指针计算， MPLAB C32 C 编译器将这些类型的长度分配为 1。

表 **1-6**：　　**-WALL** 不隐含的警告与错误选项　（续）

选项	定义
`-Wredundant-decls`	如果在同一个作用域内多次声明了任何符号则发出警告，即使多个声明都有效且没有改变任何符号。
`-Wshadow`	当一个局部变量屏蔽另一个局部变量时发出警告。
`-Wsign-compare` `-Wno-sign-compare`	当将有符号值转换为无符号值，比较有符号值和无符号值产生不正确的结果时，发出警告。该警告也可通过 `-W` 来使能。要获得 `-W` 的其他警告，而不获得这个警告，可以使用 `-W -Wno-sign-compare`。
`-Wstrict-prototypes`	如果对一个函数的定义或声明没有指定参数类型则发出警告。（如果函数定义或声明前有指定函数参数类型的声明，则允许旧式函数定义而不发出警告。）
`-Wtraditional`	如果某些语法结构在传统 C 和 ANSI C 中操作不同，产生警告。 • 宏参数出现在宏体中的字符串常量中。在传统 C 中，这些宏参数将替代参数，但在 ANSI C 中是常量的一部分。 • 在一个块中声明为 external 的函数，在块结束后被使用。 • switch 语句有 `long` 类型的操作数。 • 非静态函数声明后跟一个静态函数声明。某些传统 C 编译器不接受这种语法结构。
`-Wundef`	如果在 `#if` 伪指令中对一个未定义的标识符求值会产生警告。
`-Wunreachable-code`	如果编译器检测到代码将永远不会被执行到则发出警告。 即使在有些情况下，受影响的代码行的一部分能被执行到，该选项也可能产生警告，因此在删除明显执行不到的代码时要小心。例如，函数被内联时，警告可能表明仅在函数的一个内联拷贝中，该行执行不到。
`-Wwrite-strings`	字符串常量类型为 `const char[length]` 时，将一个字符串常量的地址复制到一个非常量 `char *` 指针会产生警告。这些警告有助于在编译时查找试图写字符串常量的代码，但仅是在声明和原型中使用 `const` 时非常小心的前提下。否则，这是不安全的，这也是 `-Wall` 为什么不要求这些警告的原因。

1.8.5 调试选项

下面列出了一些用于调试的选项。

表 1-7： 调试选项

选项	定义
-g	产生调试信息。 MPLAB C32 C 编译器支持同时使用 -g 和 -O，因此可以调试优化的代码。调试优化代码的缺点是有时可能产生异常结果： • 某些声明的变量可能根本不存在； • 控制流程可能短暂异常转移； • 某些语句可能由于计算常量结果或已经获得其值而不执行； • 某些语句可能由于被移出循环在不同的地方执行。 尽管如此，证明还是可以调试优化输出的。这使优化可能有错误的程序变得合理。
-Q	使编译器输出它在编译的每个函数名，并在结束时输出关于每遍编译的一些统计信息。
-save-temps	不要删除中间文件。将中间文件放在当前目录中，并根据源文件命名它们。因此，用 -c -save-temps 编译 foo.c 将生成下面的文件： foo.i （预处理文件） foo.s （汇编语言文件） foo.o （目标文件）

1.8.6 控制优化的选项

下面列出了一些用于控制编译器优化的选项。

表 1-8： 一般优化选项

选项	定义
-O0	不要优化。（这是默认设置。） 不指定 -O 选项，编译器的目标是降低编译成本，使调试产生期望的结果。语句是独立的：如果在语句中插入断点暂停程序，然后可以给任何一个变量赋一个新值或将程序计数器更改到指向函数中的任何其他语句，得到希望从源代码得到的结果。 编译器仅将声明为 register 的变量分配到寄存器中。
-O -O1	优化级别 1。优化编译需要花费更多的时间，且对于较大的函数，需要占用更多的存储空间。 指定 -O 选项时，编译器试图减小代码长度并缩短执行时间。 指定 -O 选项时，编译器启用 -fthread-jumps 和 -fdefer-pop，并启用 -fomit-frame-pointer。
-O2	优化级别 2。 MPLAB C32 C 编译器几乎执行所有支持的优化，而不进行空间和速度的权衡。 -O2 启用除循环展开（-funroll-loops）、函数内联（-finline-functions）及严格别名优化（-fstrict-aliasing）外的所有可选优化。该选项还启用强制复制存储器操作数 （-fforce-mem）和帧指针删除（-fomit-frame-pointer）。与 -O 相比，该选项增加了编译时间，但提高了生成代码的性能。
-O3	优化级别 3。 -O3 启用所有 -O2 指定的优化并启用内联函数选项。
-Os	优化代码长度。 -Os 使能一般不增加代码长度的所有 -O2 优化。同时执行用于减小代码长度的其他优化。

下面的选项用于控制特定的优化。 -O2 选项启用这些优化中除 -funroll-loops、-funroll-all-loops 和 -fstrict-aliasing 外的所有优化选项。

在少数情况下，当需要进行 “微调”优化时，可以使用下面的选项。

表 1-9： 特定优化选项

选项	定义
`-falign-functions` `-falign-functions=`*n*	当函数的开头对齐到下一个大于 *n* 的 2 的次幂，最多跳过 *n* 字节。例如， `-falign-functions=32` 将函数对齐到下一个 32 字节边界，但是 `-falign-functions=24` 仅在可以通过跳过等于或小于 23 字节能对齐到下一个 32 字节边界的情况下，才将函数对齐到下一个 32 字节边界。 `-fno-align-functions` 和 `-falign-functions=1` 是等价的，表明函数不会被对齐。 汇编器仅当 *n* 为 2 的次幂时，才支持这个标志；因此 *n* 是向上舍入的。如果不指定 *n*，则使用由机器决定的默认设置。
`-falign-labels` `-falign-labels=`*n*	将所有分支的目标地址对齐到 2 的次幂边界，像 `-falign-functions` 一样，最多跳过 *n* 字节。该选项可能容易使代码速度变慢，因为当以代码的通常流程到达分支的目标地址时，它必须插入空操作。 如果 `-falign-loops` 或 `-falign-jumps` 可用，并且大于这个值，则使用它们的值。 如果不指定 *n* ，则使用由机器决定的默认设置，很可能是 1，表明不对齐。
`-falign-loops` `-falign-loops=`*n*	将循环对齐到 2 的次幂边界，像 `-falign-functions` 一样，最多跳过 *n* 字节。希望循环能执行许多次，从而补偿执行的任何空操作。 如果不指定 *n*，则使用由机器决定的默认设置。
`-fcaller-saves`	通过在函数调用前后发出其他指令来保护和恢复寄存器，使能将值分配到会被函数调用破坏的寄存器中。仅当这种分配能生成更好的代码时才进行这种分配。
`-fcse-follow-jumps`	在公共子表达式消除中，当任何其他路径都不到达跳转的目标地址时，浏览跳转指令。例如，当 CSE 遇到一条带有 `else` 子句的 `if` 语句时，当条件检测为假时， CSE 将跟随跳转。
`-fcse-skip-blocks`	这与 `-fcse-follow-jumps` 类似，但使 CSE 跟随根据条件跳过块的跳转。当 CSE 遇到一个没有 `else` 子句的简单 `if` 语句时， `-fcse-skip-blocks` 使 CSE 跟随 `if` 前后的跳转。
`-fexpensive-optimizations`	执行许多成本较高的次要优化。
`-ffunction-sections` `-fdata-sections`	将每个函数或数据项放到输出文件中其自己的段。函数名或数据项名决定输出文件中的段名。 仅当使用这些选项有明显的好处时，才使用这些选项。当指定这些选项时，汇编器和链接器可能生成较大的目标文件和可执行文件，且速度较慢。
`-fgcse`	执行全局公共子表达式消除。这会同时执行全局常量和复制传播。

表 **1-9**： 特定优化选项 （续）

选项	定义
`-fgcse-lm`	使能 `-fgcse-lm` 时，全局公共子表达式消除将试图移动仅能被向其中存储破坏的装载。这允许将包含装载 / 存储序列的循环改变为循环外的装载，以及循环内的复制 / 装载。
`-fgcse-sm`	使能 `-fgcse-sm` 时，将在公共子表达式消除后运行存储移动。这试图将存储移出循环。当将该选项与 `-fgcse-lm` 一起使用时，包含装载 / 存储序列的循环可改变为循环前的装载和循环后的存储。
`-fmove-all-movables`	强制将循环内所有不可变的计算移出循环。
`-fno-defer-pop`	每次函数调用时，总是在函数一返回时就弹出函数的参数。编译器通常允许几个函数调用的参数累积在堆栈中，并将所有参数一次弹出堆栈。
`-fno-peephole` `-fno-peephole2`	禁止特定于机器的窥孔 （peephole）优化。窥孔优化发生在编译过程中的不同点。 `-fno-peephole` 禁止对机器指令进行窥孔优化，而 `-fno-peephole2` 禁止高级窥孔优化。要完全禁止窥孔优化，要同时使用这两个选项。
`-foptimize-register-move` `-fregmove`	试图重新分配 move 指令中的寄存器编号，并作为其他简单指令的操作数来增加关联的寄存器数量。 `-fregmove` 和 `-foptimize-register-moves` 是相同的优化。
`-freduce-all-givs`	强制循环中的所有一般归纳变量降低强度。 这些选项可能生成更好或更差的代码。其结果在很大程度上取决于源代码中循环的结构。
`-frename-registers`	试图通过使用寄存器分配后余下的寄存器来避免经过调度的代码中的假相关性。这种优化对于有许多寄存器的处理器比较有用。但它可能使调试无法进行，因为变量将不会存储在固定的寄存器中。
`-frerun-cse-after-loop`	在执行循环优化后，重新运行公共子表达式消除。
`-frerun-loop-opt`	运行循环优化两次。
`-fschedule-insns`	试图对指令重新排序，以消除由于所需的数据不可用而导致的指令停顿。
`-fschedule-insns2`	类似于 `-fschedule-insns`，但要求在进行寄存器分配后再执行一次指令调度。
`-fstrength-reduce`	执行降低循环强度和删除迭代变量的优化。

表 **1-9**： 特定优化选项 （续）

选项	定义

`-fstrict-aliasing`	允许编译器采用适用于被编译语言的最严格别名规则。对于 C，这根据表达式的类型进行优化。尤其是，假定一种类型的对象不会和另一种类型的对象存放在同一地址，除非类型几乎相同。例如， `unsigned int` 可引用 `int`，但不能引用 `void*` 或 `double`。字符类型可引用任何其他类型。
	特别要注意下面的代码：
	`union a_union {`
	`  int i;`
	`  double d;`
	`};`
	
	`int f() {`
	`  union a_union t;`
	`  t.d = 3.0;`
	`  return t.i;`
	`}`
	不读最后写入的联合成员，而读其他联合成员 （称为 “type-punning”）比较常见。即使对于 `-fstrict-aliasing`，如果通过联合类型访问存储器，type-punning 也是允许的。因此上面的代码可得到期望的结果。但下面的代码可能得不到期望的结果：
	`int f() {`
	`  a_union t;`
	`  int* ip;`
	`  t.d = 3.0;`
	`  ip = &t.i;`
	`  return *ip;`
	`}`

`-fthread-jumps`	执行优化，检测一个转移的目标语句是否包含另一个条件判断。如果是这样，第一个转移改变为指向第二个转移的目标语句，或紧随其后的语句，这取决于条件是真还是假。
`-funroll-loops`	执行循环展开优化。仅对在编译时或运行时其迭代次数可以确定的循环进行这种优化。 `-funroll-loops` 隐含了 `-fstrength-reduce` 和 `-frerun-cse-after-loop`。
`-funroll-all-loops`	执行循环展开优化。对于所有的循环执行这种优化，通常这种优化会使程序运行较慢。 `-funroll-all-loops` 隐含了 `-fstrength-reduce` 和 `-frerun-cse-after-loop`。

`-fflag` 形式的选项指定独立于机器的标志。大多数标志都有正的形式和负的形式。`-ffoo` 负的形式为 `-fno-foo`。在下表中，仅列出了一种形式 （非默认的形式）。

表 **1-10**： 独立于机器的优化选项

选项	定义
`-fforce-mem`	在对存储器操作数进行算术运算之前，强制将存储器操作数复制到寄存器中。这样通过使所有存储器引用可能的公共子表达式，可生成更好的代码。当它们不是公共子表达式时，指令组合应该删除单独的寄存器装载。 `-O2` 选项启用该选项。
`-finline-functions`	将所有简单的函数合并到其调用函数中。编译器直观地决定哪些函数足够简单值得这样合并。如果合并了对某个给定函数的所有调用，且函数声明为 `static`，则通常该函数本身不作为汇编代码输出。
`-finline-limit=`*n*	默认情况下， MPLAB C32 C 编译器限制可内联的函数的大小。该选项允许控制显式声明为 inline 的函数 （即用 `inline` 关键字标记的函数）的这一限制。 *n* 是可内联的函数的大小，以虚拟指令的条数为单位 （参数处理不包括在内）。 n 的默认值为 10000。增加这个值可能导致被内联的代码更多，并可能增加编译时间和存储器开销。 减小这个值通常使编译更快，更少的代码被内联 （可能程序执行速度变慢）。该选项对于使用内联的程序尤其有用。 **注：**在这里，虚拟指令代表函数大小的抽象测量。它不代表汇编指令条数，同样对于不同版本的汇编器，它的确切含义可能会有所不同。
`-fkeep-inline-functions`	即使合并了对一个给定函数的所有调用，且函数声明为 `static`，也会输出函数的一个独立的运行时可调用形式。该选项不影响 `extern` 内联函数。
`-fkeep-static-consts`	当没有启用优化时，发出声明为 static const 的变量，即使变量没有被引用。 MPLAB C32 C 编译器默认使能该选项。如果需要强制编译器检查是否引用了这个变量，而不管是否启用了优化，可以使用 `-fno-keep-static-consts` 选项。
`-fno-function-cse`	不要将函数的地址存放在寄存器中。使调用 constant 函数的每条指令显式包含函数的地址。 该选项导致生成的代码效率不高，但对于某些企图修改程序的人来说，会对不使用该选项而生成的优化程序感到束手无策。

表 1-10： 独立于机器的优化选项 （续）

选项	定义
-fno-inline	不要理会 inline 关键字。该选项通常用于使编译器不要展开任何内联函数。如果不使能优化，不会展开任何内联函数。
-fomit-frame-pointer	对于不需要帧指针的函数，不要将帧指针存放在寄存器中。这可以避免指令保护、设置和恢复帧指针。它还使一个额外的寄存器可用于许多函数。
-foptimize-sibling-calls	优化同属和尾递归调用。

1.8.7 控制预处理器的选项

下面列出了一些用于控制编译器预处理器的选项。

表 1-11： 预处理器选项

选项	定义

-A*question* (*answer*)	断言问题 *question* 的答案 *answer*，以防用预处理条件，如 #if #*question*(*answer*) 来测试问题。 -A- 禁止通常描述目标机器的标准断言。
	例如， main 的函数原型可声明如下：
	#if #environ(freestanding)
	int main(void);
	#else
	int main(int argc, char *argv[]);
	#endif
	-A 命令行选项可用于在两个原型之间进行选择。例如，为选择二者中的第一个，可使用下面的命令行选项：
	-Aenviron(freestanding)

-A -*predicate* =*answer*	取消带谓词 *predicate* 和答案 *answer* 的断言。
-A *predicate* =*answer*	进行带谓词 *predicate* 和答案 *answer* 的断言。这个形式比仍然支持的老形式 -A *predicate*(*answer*) 好，因为这个形式不使用 shell 特殊字符。
-C	告知预处理器不要舍弃注释。与 -E 选项一起使用。
-dD	告知预处理器不要按照正确的顺序将宏定义移动到输出中。
-D*macro*	将字符串 1 作为宏定义来定义宏 *macro* 。
-D*macro*=*defn*	将宏 *macro* 定义为 *defn*。在任何 -U 选项之前处理命令行中的所有 -D 选项。
-dM	告知预处理器仅输出实际上处于预处理结尾的一系列宏定义。与 -E 选项一起使用。
-dN	与 -dD 类似，不同之处在于宏参数和内容被忽略。输出中仅包括 #define name。
-fno-show-column	不要在诊断中打印列号。如果诊断被不理解列号的程序 （如 dejagnu）浏览，这可能是必要的。
-H	打印使用的每个头文件的名称以及其他正常活动。

表 **1-11：** 预处理器选项 （续）

选项	定义
`-I-`	仅对于 `#include "`*`file`*`"`，才搜索在 `-I-` 选项之前 `-I` 选项指定的任何目录。对于 `#include <`*`file`*`>`，则不搜索这些目录。 如果在 `-I-` 之后用 `-I` 选项指定了另外的目录，则对于所有 `#include` 伪指令，都搜索这些目录。（一般情况下以这种方式使用所有 `-I` 目录。） 另外， `-I-` 选项禁止将当前目录 （即当前输入文件所在的目录）作为 `#include "`*`file`*`"` 的第一个搜索目录。无法覆盖 `-I-` 的这个作用。通过 `-I.`，可以指定搜索调用编译器时为当前目录的目录。这与预处理器在默认情况下的操作不完全相同，但一般情况下都可以这样做。 `-I-` 并不禁止使用头文件的标准系统目录。因此， `-I-` 和 `-nostdinc` 是独立的。
`-I`*`dir`*	将目录 *`dir`* 添加到要在其中搜索头文件的目录列表的开头。这可用于覆盖系统头文件，替代为您自己的版本，因为在搜索系统头文件目录之前搜索这些目录。如果使用多个 `-I` 选项，则以自左向右的顺序浏览目录，最后搜索标准系统目录。
`-idirafter` *`dir`*	将目录 *`dir`* 添加到辅助包含路径中。当一个头文件在主包含路径 （`-I` 添加的路径）的任何目录中都找不到时，搜索辅助包含路径的目录。
`-imacros` *`file`*	在处理常规输入文件之前，将文件处理为输入，舍弃生成的输出。由于舍弃了由文件生成的输出， `-imacros` *`file`* 的唯一作用是使文件中定义的宏可用在主输入中。 命令行中的任何 `-D` 和 `-U` 选项始终在 `-imacros` *`file`* 之前处理，而与写这些选项的顺序无关。所有 `-include` 和 `-imacros` 选项以写这些选项时的顺序处理。
`-include` *`file`*	在处理常规输入文件之前，将文件处理为输入。实际上，首先编译文件的内容。命令行中的任何 `-D` 和 `-U` 选项始终在 `-include` *`file`* 之前处理，而与写这些选项的顺序无关。所有 `-include` 和 `-imacros` 选项以写这些选项时的顺序处理。
`-iprefix` *`prefix`*	指定 *`prefix`* 作为后面 `-iwithprefix` 选项的前缀。
`-isystem` *`dir`*	将一个目录添加到辅助包含路径的开头，将其标记为系统目录，因此可以像处理标准系统目录一样处理这个目录。
`-iwithprefix` *`dir`*	将一个目录添加到辅助包含路径中。目录名由前缀和 *`dir`* 组成，其中前缀由前面的 `-iprefix` 指定。如果没有指定前缀，将使用包含编译器安装路径的目录作为默认目录。
`-iwithprefixbefore` *`dir`*	将一个目录添加到主包含路径中。目录名由前缀和 *`dir`* 组成，这与 `-iwithprefix` 相同。

表 **1-11**： 预处理器选项 （续）

选项	定义
`-M`	告知预处理器输出适合于描述每个目标文件的相关性的 make 的规则。对于每个源文件，预处理器输出目标为该源文件目标文件名且其相关性为它使用的所有 `#include` 头文件的 make 规则。该规则可以为单行或者太长时可用 `\-newline` 来继续。规则列表打印在标准输出中，而不是打印在预处理的 C 程序中。 `-M` 隐含 `-E` （见第 **1.8.2** 节 **“控制输出类型的选项”**）。
`-MD`	与 `-M` 类似，但将相关性信息写到一个文件，编译继续进行。包含相关性信息的文件的名称与带 `.d` 扩展名的源文件名称相同。
`-MF` *file*	当与 `-M` 或 `-MM` 一起使用时，指定在其中写入相关性信息的文件。如果不给定 `-MF` 开关，预处理器将发送规则到预处理器输出发送到的地方。 当与驱动程序选项 `-MD` 或 `-MMD` 一起使用时， `-MF` 覆盖默认的相关性输出文件。
`-MG`	将缺少的头文件视为生成的文件，并假定它们位于源文件所在的目录中。如果指定了 `-MG`，那么必须也指定 `-M` 或 `-MM`。 `-MD` 或 `-MMD` 不支持 `-MG`。
`-MM`	类似于 `-M`，但输出仅涉及到用 `#include` “*file*” 包含的用户头文件。用 `#include` <*file*> 包含的系统头文件被忽略。
`-MMD`	类似于 `-MD`，但仅涉及到用户头文件，不涉及到系统头文件。

`-MP`	该选项指示 CPP 除主文件外，还要为每个相关性添加假目标，使每个不依赖于任何其他。如果删除头文件时不更新 make-file 来匹配，这些假规则将避开 make 发出的错误。
	下面是典型的输出：
	`test.o: test.c test.h`
	`test.h:`

`-MQ`	与 `-MT` 相同，但它将特定于 `make` 的任何字符用引号括起来。 `-MQ '$(objpfx)foo.o'` 得到 `$$(objpfx)foo.o: foo.c` 默认的目标自动被引号括起来，就像指定了 `-MQ` 一样。
`-MT` *target*	改变相关性生成发出的规则的目标。默认情况下， CPP 采用主输入文件的名称，包含任何路径，删除任何文件后缀 （如 `.c`），并添加平台的通常目标后缀。结果就是目标。 `-MT` 选项将目标设置为您指定的字符串。如果需要多个目标，可将它们指定为 `-MT` 的一个参数，或使用多个 `-MT` 选项。 例如： `-MT '$(objpfx)foo.o'` 可能得到 `$(objpfx)foo.o: foo.c`

表 **1-11**： 预处理器选项 （续）

选项	定义
`-nostdinc`	不要在标准系统目录中搜索头文件。仅搜索用 `-I` 选项指定的目录 （及当前目录，如果需要的话）。（关于 `-I` 的信息，请参见**第 1.8.10 节 “目录搜索选项”**。） 通过同时使用 `-nostdinc` 和 `-I-`，可将头文件搜索路径限制为仅包括显式指定的目录。
`-P`	告知预处理器不要产生 `#line` 伪指令。与 `-E` 选项一起使用 （见**第 1.8.2 节 “控制输出类型的选项”**）。
`-trigraphs`	支持 ANSI C 三字母组合。 `-ansi` 选项也有这个作用。
`-U`*`macro`*	取消宏 *`macro`*. 定义。 `-U` 选项在所有 `-D` 选项之后，但在任何 `-include` 和 `-imacros` 选项之前起作用。
`-undef`	不要预定义任何非标准宏 （包括结构标志）。

1.8.8 汇编选项

下面列出了一些用于控制汇编器操作的选项。

表 **1-12**： 汇编选项

选项	定义
`-Wa,`*`option`*	将 *`option`* 作为一个选项传递给汇编器。如果 *`option`* 中包含逗号，说明有多个选项通过逗号分隔开。

1.8.9 链接选项

如果使用了 -c、 -S 或 -E 选项中的任何一个，则链接器不会运行，且不应将目标文件名用作参数。

表 1-13： 链接选项

选项	定义
-L*dir*	将目录 *dir* 添加到命令行选项 -l 指定的在其中搜索库的目录列表中。
-l*library*	链接时搜索名为 *ibrary* 的库。 链接器在标准目录列表中搜索库，实际上是一个名为 lib*library*.a 的文件。链接器随后对这个文件的使用，就好像已经通过文件名精确指定了这个文件一样。 在命令中的何处写该选项是有所不同的，链接器按照指定库文件和目标文件的顺序来处理这些文件。因此， foo.o -lz bar.o 先搜索 foo.o，再搜索库 z，最后搜索 bar.o。如果 bar.o 引用 libz.a 中的函数，则可能不装载这些函数。 搜索的目录包括几个标准系统目录和使用 -L 指定的任何目录。 通常采用这种方法找到的文件是库文件 （其成员为目标文件的归档文件）。链接器通过浏览归档文件查找定义目前引用过但未定义的符号的成员来处理归档文件。但如果找到的文件是一个普通的目标文件，则以通常的方式链接这个文件。使用 -l 选项 （如 -lmylib）和指定文件名 （如 libmylib.a）的唯一不同之处在于， -l 按照指定搜索几个目录。 默认情况下，链接器被指示在 <install-path>\lib 中搜索 -l 选项指定的库。对于安装到默认路径的编译器，这个目录为： c:\Program Files\Microchip\MPLAB C32\lib 可使用环境变量覆盖这个操作。
-nodefaultlibs	链接时不要使用标准系统库文件。仅指定的库文件会被传递给链接器。编译器可能产生对 memcmp、 memset 和 memcpy 的调用。这些入口通常由标准编译器库中的入口解析。当指定该选项时，应通过其他某个机制来提供这些入口点。
-nostdlib	链接时不要使用标准系统启动文件或库文件。没有启动文件，仅指定的库文件会被传递给链接器。编译器可能产生对 memcmp、 memset 和 memcpy 的调用。这些入口通常由标准编译器库中的入口解析。当指定该选项时，应通过其他某个机制来提供这些入口点。
-s	从可执行文件删除所有符号表和重定位信息。
-u *symbol*	假定 *symbol* 未定义，强制链接库模块来定义这个符号。可对不同的符号多次使用 -u 来强制装载其他库模块，这样做是合法的。
-Wl,*option*	将 *option* 作为一个选项传递给链接器。如果 *option* 中包含逗号，说明有多个选项通过逗号分隔开。
-Xlinker *option*	将 *option* 作为一个选项传递给链接器。可使用该选项提供 MPLAB C32 C 编译器不知如何识别的特定系统链接器选项。

1.8.10 目录搜索选项

下面列出的选项指定编译器到哪里查找要搜索的目录和文件。

表 **1-14：** 目录搜索选项

选项	定义
-B*prefix*	该选项指定在哪里查找可执行文件、库文件、头文件和编译器本身的数据文件。 编译器驱动程序运行子程序 pic32-cpp、 pic32-cc1、 pic32-as 和 pic32-ld 中的一个或多个子程序。它将其运行的每个程序加上 *prefix* 作为前缀。 对于要运行的每个子程序，编译器驱动程序首先使用 -B 前缀 （如果存在的话）。最后，驱动程序在当前的 PATH 环境变量中搜索子程序。 有效指定目录名的 -B 前缀也适用于链接器中的库，因为编译器将这些选项翻译为链接器的 -L 选项。它们也适用于预处理器中的头文件，因为编译器将这些选项翻译为预处理器的 -isystem 选项。在这种情况下，编译器在前缀上附加 include。
-specs=*file*	为覆盖当确定哪些开关传递给 pic32-cc1、 pic32-as 和 pic32-ld 等时， pic32-gcc 驱动程序使用的默认设置，在编译器读入标准 specs 文件后处理文件。可在命令行中指定多个 -specs=*file*，以自左向右的顺序处理这些文件。

1.8.11 代码生成约定选项

`-fflag` 形式的选项指定独立于机器的标志。大多数标志都有正的形式和负的形式。`-ffoo` 负的形式为 `-fno-foo`。在下表中，仅列出了一种形式（非默认的形式）。

表 1-15： 代码生成约定选项

选项	定义

`-fargument-alias`
`-fargument-noalias`
`-fargument-noalias-global`
	指定参数之间以及参数和全局数据之间的可能关系。
	`-fargument-alias` 指定实参（形参）可互相引用，并可引用全局存储。
	`-fargument-noalias` 指定实参不能互相引用，但可引用全局存储。
	`-fargument-noalias-global` 指定实参不能互相引用，也不能引用全局存储。
	每种语言都自动使用语言标准所要求的选项。不需要自己使用这些选项。

`-fcall-saved-`*reg*	将名为 *reg* 的寄存器视为函数保存的可分配寄存器。甚至可在其中分配临时变量或跨调用有效的变量。如果函数使用了寄存器 *reg*，那么采用这种方式编译的函数将保护和恢复这个寄存器。 对帧指针或堆栈指针使用这个选项是错误的。将这个选项用于在机器执行模型中有固定重要作用的其他寄存器，将产生灾难性结果。 将这个标志用于保存函数返回值的寄存器将产生另一种灾难性结果。 所有模块中对这个标志的使用应该一致。
`-fcall-used-`*reg*	将名为 *reg* 的寄存器视为被函数调用破坏的可分配寄存器。可将这个寄存器分配给临时变量或跨调用无效的变量。采用该选项编译的函数不会保护和恢复寄存器 *reg*。 对帧指针或堆栈指针使用这个选项是错误的。将这个选项用于在机器执行模型中有固定重要作用的其他寄存器，将产生灾难性结果。 所有模块中对这个选项的使用应该一致。
`-ffixed-`*reg*	将名为 *reg* 的寄存器视为固定寄存器。生成的代码绝对不能引用它（除非作为堆栈指针、帧指针或某个其他固定的功能）。 *reg* 必须为寄存器的名称，如 `-ffixed-$0`。

表 **1-15**： 代码生成约定选项 （续）

选项	定义

`-finstrument-functions`	编译时在函数的入口和出口生成 instrumentation 调用。在函数入口之后和函数出口之前，将通过当前函数的地址及其调用地址来调用下面的 profiling 函数。
	`void __cyg_profile_func_enter (void *this_fn, void *call_site);`
	`void __cyg_profile_func_exit (void *this_fn, void *call_site);`
	第一个参数是当前函数的起始地址，可在符号表中查找到。
	profiling 函数应由用户提供。
	函数 instrumentation 要求使用帧指针。某些优化级别禁止使用帧指针。使用 `-fno-omit-frame-pointer` 将禁止这一点。
	instrumentation 也可用于在其他函数中扩展内联的函数。 profiling 调用表明从概念上来讲在哪里进入和退出内联函数。这意味着这种函数必须具有可寻址形式。如果对一个函数的所有使用都扩展内联，这会额外增加代码长度。如果要在 C 代码中使用 `extern inline`，必须提供这种函数的可寻址形式。
	可对函数指定属性 `no_instrument_function`，在这种情况下不会进行 instrumentation。

`-fno-ident`	忽略 `#ident` 伪指令。
`-fpack-struct`	将所有结构成员无缝地压缩在一起。通常不希望使用该选项，因为它使代码不是最优化的，且结构成员的偏移量与系统库不相符。
`-fpcc-struct-return`	像长值一样，将短 `struct` 和 union 值返回到存储器中，而不是返回到寄存器中。这样做效率不高，但其优点是可以使 MPLAB C32 编译的文件与其他编译器编译的文件兼容。 短结构和联合指长度和对齐都与整型匹配的结构和联合。
`-fno-short-double`	默认情况下，编译器使用与 `float` 等价的 `double` 型。该选项使得 `double` 与 `long double` 等价。如果模块通过参数传递直接或通过共享缓冲空间间接共用 double 数据，跨模块混合使用该选项可能会产生异常结果。无论使用哪个开关设置，随产品提供的库都可正常工作。
`-fshort-enums`	按照 enum 类型声明的可能值范围的需要，为其分配字节。具体来说， enum 类型等价于有足够空间的最小整型。
`-fverbose-asm` `-fno-verbose-asm`	在生成的汇编代码中加入额外的注释信息以增强可读性。 默认设置为 `-fno-verbose-asm`，将给出额外的信息，当比较两个汇编文件时有用。
`-fvolatile`	将通过指针进行的所有存储器引用视为 volatile （易变）。
`-fvolatile-global`	将对外部和全局数据项的所有存储器引用视为 volatile。使用这个开关对于 static 数据没有影响。
`-fvolatile-static`	将对 static 数据的所有存储器引用视为 volatile。

1.9 通过命令行编译单个文件

本节说明如何编译和链接单个文件。为便于讨论，假定编译器安装在 c: 驱动器的 Program Files\Microchip\MPLAB C32 目录中。因此就有下面的目录：

- c:\Program Files\Microchip\MPLAB C32\pic32mx\include——包含标准 C 头文件的目录。
- c:\Program Files\Microchip\MPLAB C32\pic32mx\include\proc——包含针对 PIC32MX 器件的头文件的目录。
- c:\Program Files\Microchip\MPLAB C32\pic32mx\lib——存放标准库文件和启动文件的库目录结构。
- c:\Program Files\Microchip\MPLAB C32\pic32mx\include\peripheral——包含 PIC32MX 外设库头文件的目录。
- c:\Program Files\Microchip\MPLAB C32\pic32mx\lib\proc——可查找到特定器件链接描述文件片段、寄存器定义文件和配置数据的目录。
- c:\Program Files\Microchip\MPLAB C32\bin——顶层工具可执行文件所在的目录。 PATH 环境变量包含该目录。

下面是一个两数相加的简单 C 程序。

使用任何文本编辑器创建下面的程序并保存为 ex1.c。

```
#include <p32xxxx.h>

unsigned int x, y, z;

unsigned int
add(unsigned int a, unsigned int b)
{
  return(a+b);
}

int
main(void)
{
  x = 2;
  y = 5;
  z = add(x,y);
  return 0;
}
```

程序的第一行包含了头文件 p32xxxx.h，这个头文件提供了该器件的所有特殊功能寄存器的定义。关于处理器头文件的更多信息，请参见**第 4 章 “低级处理器控制”**。

在 DOS 提示符下输入如下命令行来编译该程序：

```
C:\> pic32-gcc -o ex1.out ex1.c
```

命令行选项 -o ex1.out 命名输出可执行文件 （如果未指定 -o 选项，则输出文件名为 a.out）。可执行文件可加载到 MPLAB IDE 中。

如果需要 hex 文件，例如要装入器件编程器中，可以使用下面的命令：

```
C:\> pic32-bin2hex ex1.out
```

这样就生成了一个名为 ex1.hex 的 Intel hex 文件。

1.10 通过命令行编译多个文件

将 Add() 函数移到名为 add.c 的文件中来说明在一个应用程序中多个文件的使用。即：

文件 1

```
/* ex1.c */
#include <p32xxxx.h>
int main(void);
unsigned int add(unsigned int a, unsigned int b);
unsigned int x, y, z;
int main(void)
{
  x = 2;
  y = 5;
  z = Add(x,y);
  return 0;
}
```

文件 2

```
/* add.c */
#include <p32xxxx.h>
unsigned int
add(unsigned int a, unsigned int b)
{
  return(a+b);
}
```

在 DOS 提示符下输入如下命令行来编译这两个文件：

```
C:\> pic32-gcc -o ex1.out ex1.c add.c
```

这个命令编译模块 ex1.c 和 add.c。编译的模块和编译器库文件链接，并生成可执行文件 ex1.out。

注：



第 2 章 库环境

2.1 简介

本章讨论 MPLAB C32 C 库的使用。

2.2 主要内容

本章讨论的内容包括：

- 标准 I/O
- 弱函数
- “Helper”头文件
- Multilib

2.3 标准 I/O

标准输入 / 输出库函数支持两种操作模式：简单和完全。简单模式通过用于 `stdout`、`stdin` 和 `stderr` 的单字符设备上的双函数接口来支持 I/O。完全模式支持所有标准 I/O 函数。如果应用程序调用 `fopen`，那么库将使用完全模式，否则将使用简单模式。

简单模式使用 4 个函数 `_mon_puts`、 `_mon_write`、 `_mon_getc` 和 `_mon_putc` 来执行 I/O，即执行原始设备 I/O。 `_mon_getc` 的默认实现始终返回失败 （即，默认情况下，字符输入不可用）。 `_mon_putc` 的默认实现会向 UART2 写一个字符。它假定应用程序已经对 UART 执行了所有必要的初始化。 `_mon_puts` 和 `_mon_write` 的默认实现都是简单地迭代调用 `_mon_putc`。所有 4 个函数都定义为弱 （weak）函数，所以如果需要不同的功能，用户应用程序可以覆盖它们。关于这些函数的详细信息，请参见 “*MPLAB C32 C Compiler Libraries*”。

使用完全模式的应用程序必须提供标准的低级 POSIX I/O 函数 `open`、 `read`、 `write`、 `lseek` 和 `close`。未提供任何默认实现。关于这些函数的详细信息，请参见 “*MPLAB C32 C Compiler Libraries* ”（DS51685A）。

2.4 弱函数

标准库提供了低级接口的一些弱函数实现。使用该功能的用户应用程序通常会实现这些函数的更完整版本。关于特定函数的详细信息，请参见 “*MPLAB C32 C Compiler Libraries*”（DS51685A）。

如上所述，标准 I/O 库函数使用一组弱函数进行简单输出：`_mon_write`、 `_mon_putc`、 `_mon_puts` 和 `_mon_getc`。

标准启动代码 （见**第 5.7 节 “启动和初始化”**）会直接调用如下一些弱函数，并提供弱处理程序来处理引导异常和一般异常：`_on_reset`、 `_nmi_handler`、 `_bootstrap_exception_handler`、 `_general_exception_handler` 和 `_on_bootstrap`。

标准库函数 exit 在返回之前会调用弱函数 _exit。

用于信号的标准库函数 signal 和 raise 实现为弱函数，它们始终返回失败。

用于语言环境的标准库函数 setlocale 和 localeconv 实现为弱函数，不执行任何操作。

用于访问环境变量的标准库函数 getenv 实现为弱函数，始终返回 NULL。

2.5 “HELPER”头文件

2.5.1 sys/attribs.h

对于许多常用属性，提供了一些宏，以便提高用户代码可读性。

__section__(s)	应用具有段名 s 的 section 属性。
__unique_section__	应用 unique_section 属性。
__ramfunc__	将具有该属性的函数定位到 RAM 函数代码段中。
__longramfunc__	将具有该属性的函数定位到 RAM 函数代码段中，并应用 longcall 属性。
__longcall__	应用 longcall 属性。
__ISR(v,ipl)	应用优先级为 ipl 的 interrupt 属性和向量号为 v 的 vector 属性。
__ISR_AT_VECTOR(v,ipl)	应用优先级为 ipl 的 interrupt 属性和向量号为 v 的 at_vector 属性。

2.5.2 sys/kmem.h

系统代码可能需要在虚拟和物理地址之间，以及在内核段地址之间转换。提供了一些宏，帮助更方便地进行这些转换和确定地址所处的段。

KVA_TO_PA(v)	将内核虚拟地址转换为物理地址。
PA_TO_KVA0(pa)	将物理地址转换为 KSEG0 虚拟地址。
PA_TO_KVA1(pa)	将物理地址转换为 KSEG1 虚拟地址。
KVA0_TO_KVA1(v)	将 KSEG0 虚拟地址转换为 KSEG1 虚拟地址。
KVA1_TO_KVA0(v)	将 KSEG1 虚拟地址转换为 KSEG0 虚拟地址。
IS_KVA(v)	如果地址是内核段虚拟地址，则值为 1，否则值为 0。
IS_KVA0(v)	如果地址是 KSEG0 虚拟地址，则值为 1，否则值为 0。
IS_KVA1(v)	如果地址是 KSEG1 虚拟地址，则值为 1，否则值为 0。
IS_KVA01(v)	如果地址是 KSEG0 或 KSEG1 虚拟地址，则值为 1，否则值为 0。

2.6 MULTILIB

2.6.1 什么是 Multilib？

使用 multilib，目标库将使用一组置换选项构建多次。 Multilib 是使用这些选项进行构建所产生的一组目标库。当调用编译器 shell 来编译并链接应用程序时， shell 会选择使用相同选项构建的目标库的版本。

2.6.2 有哪些 Multilib 可供 MPLAB C32 语言工具使用？

随 MPLAB C32 C 编译器分配的目标库使用以下选项进行构建：

- 长度与速度 （-Os 与 -O3）
- 16 位与 32 位 （-mips16 与 -mno-mips16）
- 软件浮点与无浮点支持 （-msoft-float 与 -mno-float）

默认情况下， MPLAB C32 语言工具使用 -O0、 -mno-mips16 和 -msoft-float 进行编译。因此，需要关心的选项为 -Os 或 -O3、 -mips16 和 -mno-float。提供了使用以下命令行选项构建的库：

1. 默认命令行选项
2. -Os
3. -O3
4. -mips16
5. -mno-float
6. -mips16 -mno-float
7. -Os -mips16
8. -Os -mno-float
9. -Os -mips16 -mno-float
10. -O3 -mips16
11. -O3 -mno-float
12. -O3 -mips16 -mno-float

2.6.3 Multilib 目录的位置？

默认情况下， MPLAB C32 语言工具使用目录 <install-directory>/lib/gcc/ 来存储特定库，使用目录 <install-directory>/<pic32mx>/lib 来存储特定于目标的库。这两个目录结构中均包含了针对以上指定的每个 multilib 组合的子目录。这些子目录分别如下：

1. .
2. ./size
3. ./speed
4. ./mips16
5. ./no-float
6. ./mips16/no-float
7. ./size/mips16
8. ./size/no-float
9. ./size/mips16/no-float
10. ./speed/mips16
11. ./speed/no-float
12. ./speed/mips16/no-float

2.6.4 选择哪一个 Multilib 目录？

本节提供了一些示例，以及关于选择哪一个 multilib 子目录的详细信息。

1. `pic32-gcc foo.c`

   对于该示例，未指定任何命令行选项 （即，使用默认的命令行选项）。这种情况下，将使用 `.` 子目录。

2. `pic32-gcc -Os foo.c`

   对于该示例，指定了用于优化大小的命令行选项 （即，使用了 `-Os`）。这种情况下，将使用 `./size` 子目录。

3. `pic32-gcc -O2 foo.c`

   对于该示例，指定了进行优化的命令行选项，但是，该命令行选项既不优化大小也不优化空间 （即，使用了 `-O2`）。这种情况下，将使用 `.` 子目录。

4. `pic32-gcc -Os -mips16 foo.c`

   对于该示例，指定了优化大小和 MIPS16 代码的命令行选项 （即，使用了 `-Os` 和 `-mips16`）。这种情况下，将使用 `./size/mips16` 子目录。



第 3 章 中断

3.1 简介

中断处理对于大多数单片机应用来说都是很重要的一个方面。中断用来使软件操作与实时发生的事件同步。当发生中断时，软件的正常执行流程被打断，调用专门的函数来处理事件。当中断处理结束时，恢复先前的现场信息并继续正常执行流程。

PIC32MX 器件支持多个内部和外部中断源。另外，允许高优先级中断中断任何正在处理的低优先级中断。

MPLAB C32 C 编译器完全支持在 C 或行内汇编代码中进行中断处理。本章将对中断处理做一个概括介绍。

3.2 主要内容

本章讨论的内容包括：

- 指定中断处理程序函数
- 将中断处理程序函数与异常向量相关联
- 异常处理程序

3.3 指定中断处理函数

中断处理函数用于实现现场保护和恢复，以确保从中断返回时，程序现场恢复进入中断前的状态。

3.3.1 处理程序函数现场保护

C 函数的标准调用约定始终会保留 `zero`、 `s0-s7`、 `gp`、 `sp` 和 `fp`。 `k0` 和 `k1` 供编译器访问和保留非 GPR 现场，但对它们的存取始终是以原子方式进行的 （即，在禁止全局中断的序列中），所以不需要主动保存它们。除了标准寄存器之外，处理程序函数还会主动保存 `a0-a3`、 `t0-t9`、 `v0`、 `v1` 和 `ra` 寄存器。

中断处理函数还会主动保存和恢复处理程序函数所使用的处理器状态寄存器。特别是， `EPC`、 `SR`、 `hi` 和 `lo` 寄存器会被保留为现场。

优先级指定为 7 （最高优先级）的处理程序函数将使用影子寄存器组来保留通用寄存器，使得进入处理程序函数的应用程序代码的延时可以较短。

3.3.2 将一个函数标记为中断处理程序

函数通过 interrupt 属性或中断 pragma[1] 伪指令标记为处理程序函数。每种方法在功能上是彼此等价的。中断可以指定为具有特定优先级的处理中断，也可以指定为在单向量模式下工作。

1. 请注意，预处理器宏在 pragma 伪指令中不会展开。

```
# pragma interrupt function-name ipln [vector [@]vector-number [,
vector-number-list]]
# pragma interrupt function-name  single [vector [@] 0
```

其中， *n* 处于范围 0..7 内 （且包括 0 和 7）。 `iplx` 说明符可以全为大写或全为小写。

使用中断 pragma 伪指令指示的处理程序函数的函数定义必须紧跟在 pragma 伪指令之后且与其位于同一个翻译单元中。

`interrupt` 属性也指示函数定义是中断处理程序。它在功能上等价于中断 pragma 伪指令。

例如，以下两个 `foo` 定义均指示它是优先级为 4 的中断处理函数。

```
#pragma interrupt foo ipl4
void foo (void)
```

在功能上等价于

```
void __attribute__ ((interrupt(ipl4))) foo (void)
```

3.4 将中断处理函数与异常向量相关联

共有 64 个异常向量，编号为 0 至 63。每个中断源都按器件数据手册中的规定映射到一个异常向量。默认情况下，在每个向量地址处会保留 4 个字的空间，用于将中断分派给该异常源的处理程序函数。

中断处理函数可以通过以下方式与中断向量相关联：指定为位于异常向量地址处的分派函数的目标，或者直接定位到异常向量地址处。单个处理程序函数可以作为多个分派函数的目标。

处理程序函数与一个或多个异常向量地址之间的关联可以通过中断 pragma 伪指令的子句、单独的向量 pragma 伪指令或函数声明中的向量属性指定。

3.4.1 中断 Pragma 子句

中断 pragma 伪指令具有可选的 `vector` 子句，跟随在优先级说明符之后。

```
# pragma interrupt function-name ipl-specifier [vector
[@]vector-number [, vector-number-list]]
```

目标为指定处理程序函数的分派函数将创建在指定向量号的异常向量地址处。如果所指定的第一个向量号前面带有 “@” 符号，那么处理程序函数本身将直接定位到该位置。

例如，以下 pragma 伪指令指定函数 `foo` 将创建为优先级为 4 的中断处理函数。 `foo` 将定位到异常向量 54 的地址处。目标为 `foo` 的分派函数将创建在异常向量 34 的地址处。

```
#pragma interrupt foo ipl4 vector @54, 34
```

以下 pragma 伪指令指定函数 `bar` 将创建为优先级为 5 的中断处理函数。 `bar` 将定位到通用程序存储区 （.text 段）中。在异常向量 23 所在的地址处将创建目标为 `bar` 的分派函数。

```
#pragma interrupt bar ipl5 vector 23
```


3.4.2 Vector Pragma

vector pragma 用于创建一个或多个目标为所指示函数的分派函数。对于使用 interrupt pragma 指定的目标函数，它的作用就如同是使用了 vector 子句。vector pragma 的目标函数可以是任意函数，包括以汇编或其他方式实现的外部函数。

```
# pragma vector function-name vector vector-number [,
vector-number-list]
```

以下 pragma 伪指令定义了一个分派函数，它的目标是处于异常向量 54 所在的地址处的函数 foo。

```
#pragma vector foo 54
```

3.4.3 Vector 属性

处理程序函数可以通过属性与一个或多个异常向量地址相关联。 at_vector 属性指示应将处理程序函数本身放在异常向量地址处。 vector 属性指示应在异常向量地址处创建分派函数。

例如，以下声明指定函数 foo 将创建为优先级为 4 的中断程序函数。 foo 将定位到异常向量 54 的地址处。

```
void __attribute__ ((interrupt(ipl4))) __attribute__ ((at_vector(54)))
foo (void)
```

以下声明指定函数 foo 将创建为优先级为 4 的中断处理函数。并在异常向量 52 和 53 所在的地址处定义目标为 foo 的分派函数。

```
void __attribute__ ((interrupt(ipl4))) __attribute__ ((vector(53,
52))) foo (void)
```

3.5 异常处理程序

PIC32MX 器件还有两个用于非中断异常的异常向量。这些异常归类为引导异常和一般异常。

3.5.1 引导异常

复位异常是在引导代码运行时 （Status_{BEV}=1）发生的任何异常。所有复位异常的向量地址都是 0xBFC00380。

在该地址单元， MPLAB C32 工具链会放置一条转移指令，其目标为名为 _bootstrap_exception_handler() 的函数。在标准库中，提供了该函数的默认弱版本，它只是进入一个无限循环。如果用户应用程序提供了 _bootstrap_exception_handler() 的实现，那么将改为使用该实现版本。

3.5.2 一般异常

一般异常是引导代码之外的程序执行期间 （Status_{BEV}=0）发生的任何非中断异常。一般异常向量的地址对于 EBase 的偏移量为 0x180。

在该地址单元， MPLAB C32 工具链会放置一条转移指令，其目标为名为 _general_exception_context() 的函数。所提供的该函数的实现版本会保存现场、调用应用程序处理程序函数、恢复现场，以及执行从异常返回的指令。所保存的现场是 hi 和 lo 寄存器，以及除 s0-s8 之外的所有通用寄存器， s0-s8 定义为由所有被调用函数进行保留，所以不需要在此处再次主动保存。 Cause 和 Status 寄存器的值会被传递给应用程序处理函数 （_general_exception_handler()）。如果用户应用程序提供了 _general_exception_context() 的实现，那么将改为使用该实现版本。

```
void _general_exception_handler (unsigned cause, unsigned status);
```

标准库中提供了 _general_exception_handler() 弱版本的默认实现，该实现只是进入一个无限循环。如果用户应用程序提供了 _general_exception_handler() 的实现，那么将改为使用该实现版本。



第 4 章 低级处理器控制

4.1 简介

本章讨论 PIC32MX 器件的低级寄存器和配置的访问。

4.2 主要内容

本章讨论的内容包括：

- 通用处理器头文件
- 处理器支持头文件
- 外设库函数
- 特殊功能寄存器访问
- CP0 寄存器访问
- 配置位访问

4.3 通用处理器头文件

通用处理器头文件是 C 文件，它根据使用 `-mprocessor` 命令行选项指定的处理器而包含正确的特定于处理器的头文件。通用处理器头文件位于 `c:\Program Files\Microchip\MPLAB C32\pic32mx\include`；其中，`c:\Program Files\Microchip\MPLAB C32` 是 MPLAB C32 工具链的安装目录。除了包含正确的特定于处理器的头文件之外，通用处理器头文件还提供了 `#define` 伪指令，使得可以使用来自汇编语言文件的约定寄存器名称。

要包含通用处理器头文件，请在源代码中使用以下语句：

```
#include <p32xxxx.h>
```

包含通用处理器头文件使得可以为 MPLAB C32 工具链所支持的任意处理器编译源代码，而无需更改要包含的文件。

4.4 处理器支持头文件

特定于处理器的头文件是一些包含了在 C 或汇编语言中使用的特殊功能寄存器（Special Function Register， SFR）的外部声明的文件。依照约定，每个 SFR 都使用数据手册中的相同名称进行命名——例如， `WDTCON` 代表看门狗定时器控制寄存器。如果寄存器含有可能关注的一些个别位，那么还会有一个为该 SFR 定义的结构 `typedef`；其中，结构 `typedef` 的名称是附加了 `bits_t` 的寄存器的名称——例如， `__WDTCONbits_t`。这些个别位（或位域）在结构中使用数据手册中的名称进行命名。例如，在 PIC32MX360F512L 的特定于处理器的头文件中，供 C 使用的 `WDTCON` 寄存器声明为：

```
extern volatile unsigned int WDTCON __attribute__((section("sfrs")));
typedef union {
  struct {
    unsigned WDTCLR:1;
    unsigned :1;
    unsigned SWDTPS0:1;
    unsigned SWDTPS1:1;
    unsigned SWDTPS2:1;
    unsigned SWDTPS3:1;
    unsigned SWDTPS4:1;
    unsigned :8;
    unsigned ON:1;
  };
  struct {
    unsigned :2;
    unsigned WDTPSTA:5;
    unsigned :1;
    unsigned PWRTPSTA:3;
  };
  struct {
    unsigned w:32;
  };

} __WDTCONbits_t;
extern volatile __WDTCONbits_t WDTCONbits asm ("WDTCON") __attribute__((section("sfrs")));
```

注：	符号 `WDTCON` 和 `WDTCONbits` 指代相同的寄存器，解析为相同的地址，这从 `WDTCONbits` 的声明就可以看出。

用于汇编语言时， `WDTCON` 寄存器声明为：`.extern WDTCON`。

特定于处理器的头文件位于 `c:\Program Files\Microchip\MPLAB C32\pic32mx\include\proc`；其中，`c:\Program Files\Microchip\MPLAB C32` 是 MPLAB C32 工具链的安装目录。要包含特定于处理器的头文件，建议您包含通用处理器头文件 （见**第 4.3 节 "通用处理器头文件"**）；但是，如果您希望特别调用特定于处理器的头文件，请在源文件中使用以下语句 （示例假定是为 PIC32MX360F512L 包含特定于处理器的头文件）：

```
#include <proc/p32mx360f512l.h>
```

4.5 外设库函数

随编译器工具提供的外设库函数支持 PIC32MX 器件的许多外设。关于所提供的函数的详细信息，请参见 《MPLAB® C32 C 编译器函数库》（DS51685A_CN）。

4.6 特殊功能寄存器访问

在应用程序中使用 SFR 时，需要执行 3 个步骤。

1. 包含通用处理器头文件 （即， p32xxxx.h）或相应器件的特定于处理器的头文件 （例如， proc/p32mx360f512l.h）。
   ```
   #include <p32xxxx.h>
   ```
2. 像访问任何其他 C 变量一样访问 SFR。源代码可以写和 / 或读 SFR。例如，以下语句将 Timer1 的特殊功能寄存器中的所有位清零：
   ```
   TMR1 = 0;
   ```
   下一条语句使能看门狗定时器：
   ```
   WDTCONbits.ON = 1;
   ```
3. 使用默认链接描述文件进行链接，或者在项目中包含相应处理器的 processor.o 文件。

4.7 CP0 寄存器访问

4.7.1 CP0 寄存器定义头文件

CP0 寄存器定义头文件 （cp0defs.h）是包含 CP0 寄存器及其位域的定义的文件。此外，它还包含了用于访问 CP0 寄存器的宏。 CP0 寄存器定义头文件位于 c:\Program Files\Microchip\MPLAB C32\pic32mx\include ；其中， c:\Program Files\Microchip\MPLAB C32 是 MPLAB C32 工具链的安装目录。CP0 寄存器定义头文件旨在与汇编或 C 文件配合使用。

CP0 寄存器定义头文件依赖于在处理器通用头文件中定义的宏 （见**第 4.3 节 “通用处理器头文件”**）。要包含 CP0 寄存器定义头文件，请在源代码中使用以下语句：

```
#include <p32xxxx.h>
```

4.7.2 CP0 寄存器定义

当从汇编文件中包含 CP0 寄存器定义头文件时， CP0 寄存器定义为：

```
#define _CP0_REGISTER_NAME $register_number, select_number
```

例如， IntCtl 寄存器定义为：

```
#define _CP0_INTCTL $12, 1
```

当从 C 文件中包含 CP0 寄存器定义头文件时， CP0 寄存器和选择定义为：

```
#define _CP0_REGISTER_NAME register_number
#define _CP0_REGISTER_NAME_SELECT select_number
```

例如， IntCtl 寄存器定义为：

```
#define _CP0_INTCTL 12
#define _CP0_INTCTL_SELECT 1
```

4.7.3 CP0 寄存器位域定义

当从汇编或 C 文件中包含 CP0 寄存器定义头文件时，对于每个 CP0 寄存器位域，存在 3 个 #define。

_CP0_*REGISTER_NAME_FIELD_NAME*_POSITION——起始位地址

_CP0_*REGISTER_NAME_FIELD_NAME*_MASK——属于该位域的位置 1

_CP0_*REGISTER_NAME_FIELD_NAME*_LENGTH——该位域占用的位数

例如， IntCtl 寄存器的向量间隔位域具有以下定义：

```
#define _CP0_INTCTL_VS_POSITION 0x00000005
#define _CP0_INTCTL_VS_MASK     0x000003E0
#define _CP0_INTCTL_VS_LENGTH   0x00000005
```

4.7.4 CP0 访问宏

当从 C 文件中包含 CP0 寄存器定义头文件时，即定义了 CP0 访问宏。每个 CP0 寄存器最多可以定义 6 个不同的访问宏：

_CP0_GET_*REGISTER_NAME* ()	返回寄存器 *REGISTER_NAME* 的值。
_CP0_SET_*REGISTER_NAME* (val)	将寄存器 *REGISTER_NAME* 设置为 val，并返回 void。仅用于定义包含可写位域的寄存器。
_CP0_XCH_*REGISTER_NAME* (val)	将寄存器 *REGISTER_NAME* 设置为 val，并返回原先的寄存器值。仅用于定义包含可写位域的寄存器。
_CP0_BIS_*REGISTER_NAME* (set)	将寄存器 *REGISTER_NAME* 设置为（reg |= set），并返回原先的寄存器值。仅用于定义包含可写位域的寄存器。
_CP0_BIC_*REGISTER_NAME* (clr)	将寄存器 *REGISTER_NAME* 设置为（reg &= ~clr），并返回原先的寄存器值。仅用于定义包含可写位域的寄存器。
_CP0_BCS_*REGISTER_NAME* (clr, set)	将寄存器 *REGISTER_NAME* 设置为（reg = (reg & ~clr) | set），并返回原先的寄存器值。仅用于定义包含可写位域的寄存器。

4.8 配置位访问

4.8.1 #pragma config

#pragma config 伪指令指定要由应用程序使用的特定于处理器的配置设置（即，配置位）。更多信息，请参见 “*PIC32MX Configuration Settings*” 在线帮助。

配置设置可以使用多个 #pragma config 伪指令指定。 MPLAB C32 C 编译器会验证所指定的配置设置对于它进行编译所针对的处理器是否有效。如果配置字中的给定设置未在任何 #pragma config 伪指令中指定，那么与那些设置关联的位默认为未设定值。

对于使用 #pragma config 伪指令指定了设置的每个配置字，编译器会生成名为 .config_*address* 的只读数据段；其中， *address* 是配置字地址的十六进制表示。例如，如果为位于地址 0xBFC02FFC 的配置字指定了配置设置，那么会创建名为 .config_BFC02FFC 的只读数据段。

4.8.1.1 语法

pragma-config 伪指令：

      # pragma config *setting-list*

setting-list:

    setting

  | *setting-list*, *setting*

setting:

    setting-name **=** *value-name*

setting-name 和 *value-name* 是特定于器件的名称，可以使用 *"PIC32MX Configuration Settings"* 文档来确定。

4.8.1.2 示例

以下示例说明可以如何使用 `#pragma config` 伪指令。示例执行以下操作：

- 使能看门狗定时器，
- 将看门狗后分频比设置为 1:128，并
- 选择 HS 振荡器作为主振荡器

```
#pragma config FWDTEN = ON, WDTPS = PS128
#pragma config POSCMOD = HS
...
void main (void)
{
...
}
```

注：



第 5 章 编译器运行时环境

5.1 简介

本章讨论 MPLAB C32 C 编译器的运行时环境。

5.2 主要内容

本章讨论的内容包括：

- 寄存器约定
- 堆栈使用
- 堆使用
- 函数调用约定
- 启动和初始化
- 默认链接描述文件的内容
- RAM 函数

5.3 寄存器约定

表 5-1： 寄存器约定

寄存器名称	软件名称	使用
$0	zero	读取时始终为 0。
$1	at	汇编器临时变量。
$2-$3	v0-v1	函数的返回值。
$4-$7	a0-a3	用于向函数传递参数。
$8-$15	t0-t7	临时寄存器，供编译器对表达式求值。调用函数时，不会保存这些寄存器的值。
$16-$23	s0-s7	临时寄存器，调用函数时，会保存这些寄存器的值。
$24-$25	t8-t9	临时寄存器，供编译器对表达式求值。调用函数时，不会保存这些寄存器的值。
$26-$27	k0-k1	保留供中断 / 陷阱处理程序使用。
$28	gp	全局指针。
$29	sp	堆栈指针。
$30	fp 或 s8	在需要帧指针时作为帧指针。在不需要时作为附加的临时保存的寄存器。
$31	ra	函数的返回地址。

5.4 堆栈使用

MPLAB C32 C 编译器使用通用寄存器 29 专门作为软件堆栈指针。所有的处理器堆栈操作 （包括函数调用、中断和异常）都使用软件堆栈。堆栈从高地址到低地址向下增长。

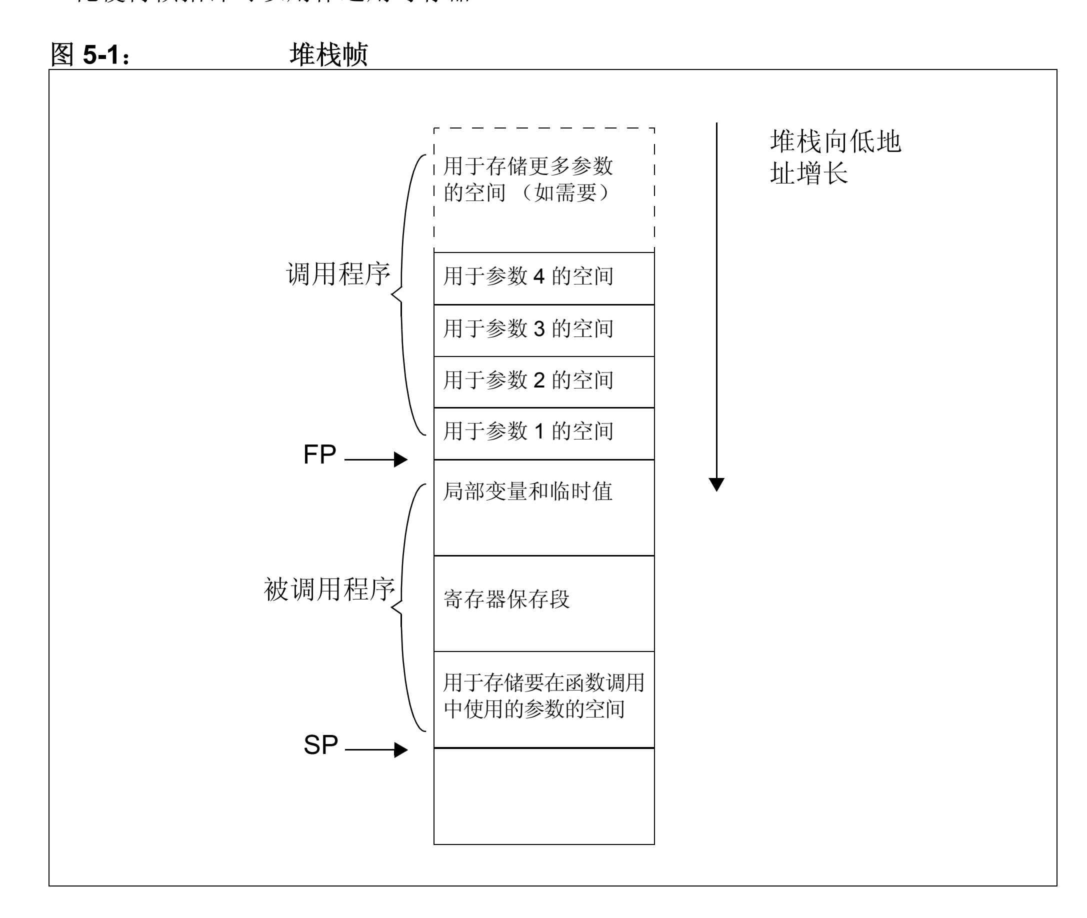
默认情况下，堆栈的大小为 1024 字节。堆栈大小可以通过在链接器命令行中使用 `--defsym_min_stack_size` 链接器命令行选项指定大小进行更改。以下是使用命令行分配 2048 字节的堆栈的示例：

```
pic32-gcc foo.c -Wl,--defsym,_min_stack_size=2048
```

运行时堆栈从高地址到低地址向下增长 （见图 5-1）。编译器使用两个工作寄存器来管理堆栈：

- 寄存器 29 （`sp`）——这是堆栈指针。它指向堆栈中的下一个可用单元。
- 寄存器 30 （`fp`）——这是帧指针。它指向当前函数的帧。在需要时，每个函数会创建一个新帧，通过该帧分配自动变量和临时变量。编译器优化可能会取消通过帧指针进行的堆栈指针引用，而将它们转换为通过堆栈指针进行的等效引用。这种优化使得帧指针可以用作通用寄存器。

图 5-1： 堆栈帧



5.5 堆使用

C 运行时堆是数据存储器中的未初始化段，用于使用标准 C 函数库中的动态存储器管理函数 calloc、 malloc 和 realloc 进行动态存储器分配。如果不使用以上函数，那么就不需要分配堆。默认情况下，不会创建堆。

如果希望使用动态存储器分配，无论是通过调用一个存储器分配函数直接使用，还是通过使用以上函数的标准 C 库函数间接使用，那么必须创建一个堆。堆通过在链接器命令行中使用 --defsym_min_heap_size 链接器命令行选项指定其大小来创建。以下是使用命令行分配 512 字节的堆的示例：

```
pic32-gcc foo.c -Wl,--defsym,_min_heap_size=512
```

链接器在紧接堆栈前面的位置分配堆。

5.6 函数调用约定

堆栈指针始终在 4 字节边界处对齐。

- 所有长度小于32位的整数类型首先会被转换为32位的值。参数的前4个32位通过寄存器 a0-a3 传递 （关于每种数据类型需要多少个寄存器，请参见表 5-2）。
- 虽然一些参数可能通过寄存器传递，但还是会在堆栈中为要传递给函数的所有参数分配空间 （见图 5-2）。
- 调用函数时：
  - 寄存器 a0-a3 用于向函数传递参数。调用函数时，不会保留这些寄存器的值。
  - 寄存器 t0-t7 和 t8-t9 是由调用程序保存的寄存器。调用函数必须将这些值压入堆栈，以保存寄存器的值。
  - 寄存器 s0-s7 是由被调用程序保存的寄存器。被调用函数必须保存这些寄存器中会被修改的寄存器的值。
  - 如果优化器取消将寄存器 s8 用作帧指针，那么该寄存器的值需要保存。否则， s8 是被保留的寄存器。
  - 寄存器 ra 中包含函数调用的返回地址。

表 5-2： 需要的寄存器

数据类型	需要的寄存器数
char	1
short	1
int	1
long	1
long long	2
float	1
double	2
long double	2
Structure	最大为 4 （取决于结构的大小）。

图 5-2： 传递参数

示例 1：

```
int add (int, int)
a= add (5, 10);
```

	栈		寄存器
SP + 4	未定义	a0	5
SP	未定义	a1	10

示例 2：

```
void foo (double, double)
call= foo (10.5, 20.1);
```

	栈		寄存器
SP + 12 SP + 8	未定义	a0 a1	10.5
SP + 4 SP	未定义	a2 a3	20.1

示例 3：

```
void calculate (double, double, int)
calculate (50.3, 100.0, .10);
```

	栈		寄存器
SP + 16	.10		
SP + 12 SP + 8	未定义	a0 a1	50.3
SP + 4 SP	未定义	a2 a3	100.0

5.7 启动和初始化

5.7.1 规定

对于运行时模型，存在以下规定：

- 仅内核模式
- 仅 KSEG1
- RAM 函数带有属性 `__ramfunc__` 或 `__longramfunc__`，表示所有 RAM 函数均终止于 `.ramfunc` 段

5.7.2 PIC32MX 启动代码

PIC32MX 启动代码必须执行以下操作：

1. 在发生 NMI 时跳转到 NMI 处理程序
2. 初始化堆栈指针和堆
3. 初始化全局指针
4. 调用 "复位时" 过程
5. 清零未初始化数据段
6. 将初始化数据从程序闪存复制到数据存储器
7. 将 RAM 函数从程序闪存复制到数据存储器
8. 初始化总线矩阵寄存器
9. 初始化 CP0 寄存器
10. 跟踪控制 2 寄存器 （`TraceControl2`——CP0 寄存器 23，选择 2）
11. 调用 "引导时" 过程
12. 更改异常向量的位置
13. 调用主程序

5.7.2.1 在发生 NMI 时跳转到 NMI 处理程序

如果 NMI 导致进入复位向量，那么会跳转到 NMI 处理程序 （`_nmi_handler`）。代码中提供了弱 （weak）版本的 NMI 处理程序，它执行一条 ERET 指令。`_nmi_handler` 函数必须带有属性 nomips16， [例如，`__attribute__((nomips16))`]，因为启动代码会跳转到该函数。

5.7.2.2 初始化堆栈指针和堆

堆栈指针 （`sp`）寄存器必须在启动代码中初始化。要使启动代码能够初始化 `sp` 寄存器，链接描述文件必须初始化一个指向 KSEG1 数据存储器[1] 末尾位置的变量。该变量名为 `_stack`。用户可以通过向链接器提供命令行选项 `--defsym _min_stack_size=`*N* 来更改所分配的最小堆栈空间量。链接描述文件提供的 `_min_stack_size` 的默认值为 `1024`。

类似地，用户可能会希望在应用程序中使用堆。虽然启动代码不需要初始化堆，但是必须让标准 C 函数库 （`sbrk`）知道堆的位置及大小。链接描述文件会创建一个变量来标识堆的开始位置。堆位于所使用的 KSEG1 数据存储器的末尾位置。该变量名为 `_heap`。用户可以通过向链接器提供命令行选项 `--defsym _min_heap_size=`*M* 来更改所分配的最小堆空间量。链接描述文件提供的 `_min_heap_size` 的默认值为 `0`。如果在堆大小设置为 0 时使用堆，那么产生的行为与堆使用量超出最小堆大小时相同。即，它溢出到为堆栈分配的空间中。

1. 根据是否存在 RAM 函数，数据存储器末尾位置会有不同。如果存在 RAM 函数，那么必须对 DRM 的一部分进行配置，以使内核程序包含 RAM 函数，并且堆栈指针定位到 DRM 内核程序边界地址开始位置前一个字的位置处。如果不存在 RAM 函数，那么堆栈指针定位到 DRM 的真正末尾位置处。

堆和堆栈使用未被分配的 KSEG1 数据存储器，堆从已分配的 KSEG1 数据存储器末尾位置开始，向堆栈方向向上增长，而堆栈则从 KSEG1 数据存储器的末尾位置开始，向堆方向向下增长。如果根据所请求的最小堆大小和最小堆栈大小，没有足够空间可用，那么链接器会报告一个错误。

图 5-3：　　堆栈和堆布局

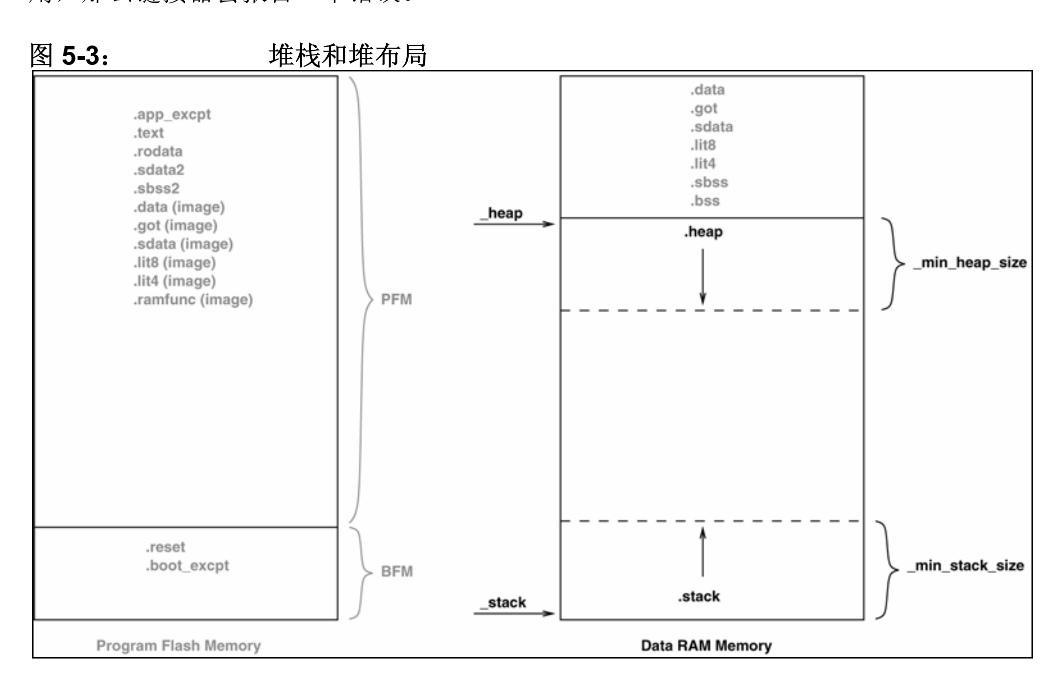
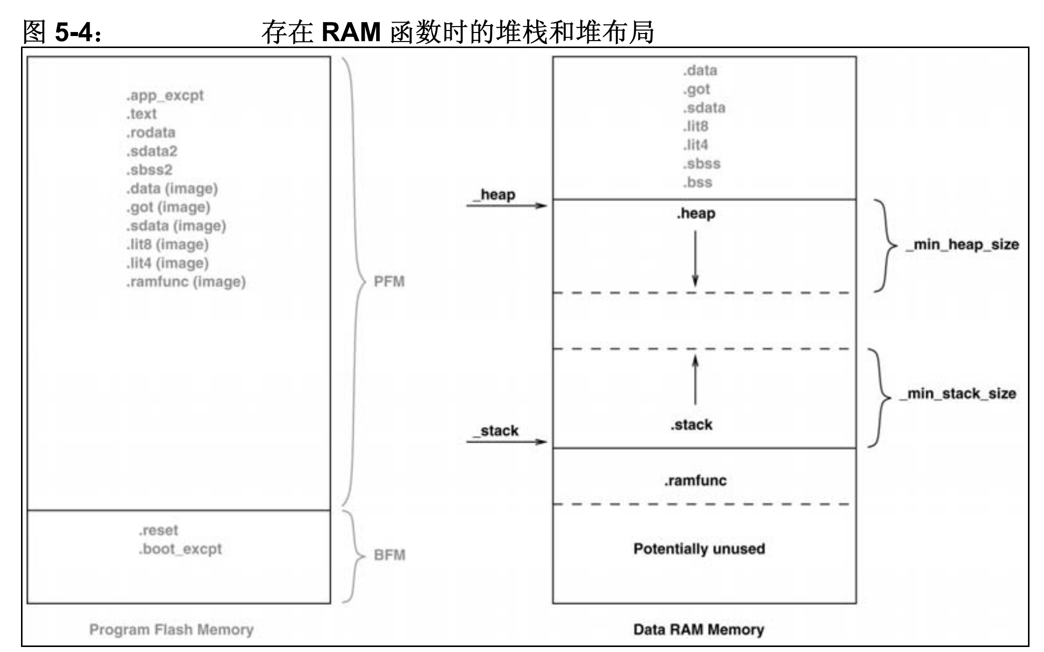


图 5-4：　　存在 RAM 函数时的堆栈和堆布局



5.7.2.3　初始化全局指针

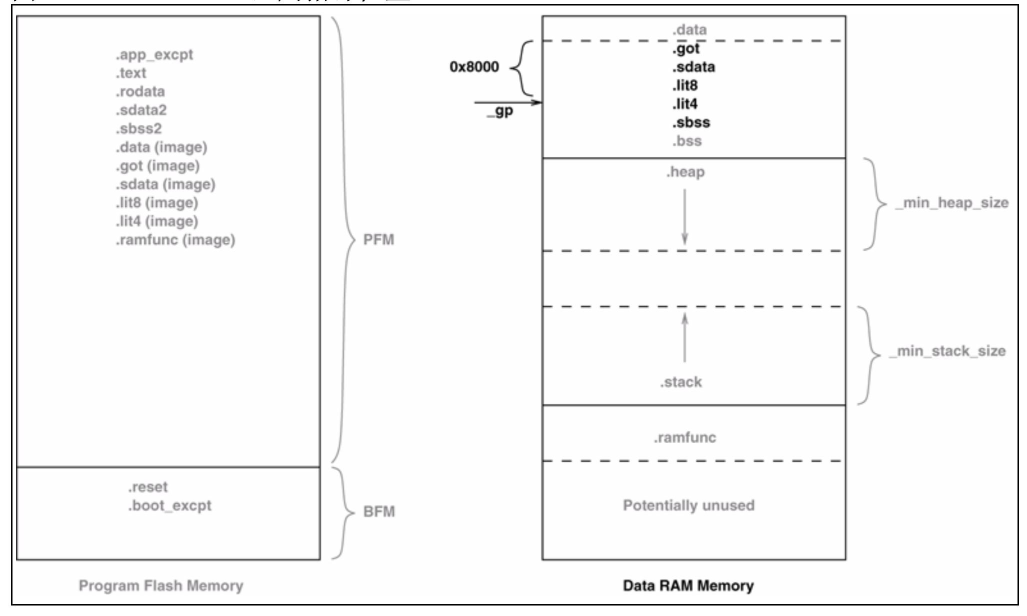
编译器工具链支持全局指针 （gp）相对寻址。在位于 gp 寄存器中存储的地址任意一侧的 32 KB 地址范围内装入或存储数据时，可以使用 gp 寄存器作为基址寄存器，在单条指令中执行该操作。在不使用全局指针的情况下，从静态存储区装入数据需要两条指令——一条指令用于装入由编译器 / 链接器计算得到的 32 位常量地址的高位，另一条指令用于装入数据。

要使用 gp 相对寻址，编译器和汇编器必须将所有 “小” 变量和常量组合到以下某个段中：

- `.lit4.`
- `.sdata.`
- `.sdata.*`
- `.gnu.linkonce.s.*`
- `lit8`
- `sbss`
- `sbss.*`
- `.gnu.linkonce.sb.*`

然后，链接器必须将以上所有输入段组合在一起。运行时启动代码必须初始化 gp 寄存器，使之指向该输出段的 “中间”。要使启动代码能够初始化 gp 寄存器，链接描述文件必须初始化一个变量，它的地址距离包含 “小” 变量和常量的输出段的开始位置 32 KB。该变量名为 `_gp` （与内核链接描述文件相匹配）。除了在标准 GPR 组中进行初始化之外，全局指针还必须在影子寄存器组中初始化。

图 5-5： 全局指针位置



5.7.2.4 调用 “复位时” 过程

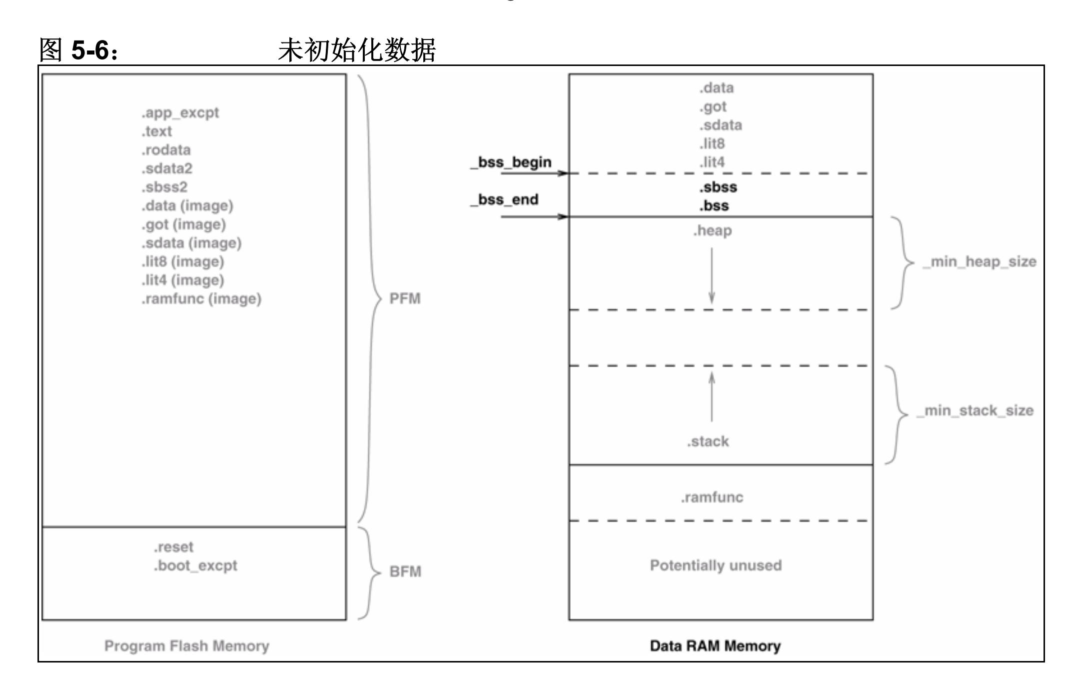
初始化最小的 “C” 现场之后，将会调用一个过程。该过程使用户可以在器件复位时立即执行一些操作。随启动代码一起，提供了该过程 （`_on_reset`）的弱版本，其内容为空。如果是以 “C” 编写该过程，那么用户需要注意一些特殊事项。最重要的是，静态分配的变量不会被初始化 （需要使用指定的初始化器，或像对待未初始化变量那样赋 0 值）。

5.7.2.5 清零未初始化数据段

存在两个未初始化数据段——`.sbss` 和 `.bss`。 `.sbss` 段是一个数据段，包含长度小于等于 *n* 字节的未初始化变量，其中的 *n* 由 `-G`*n* 命令行选项决定。 `.bss` 段是一个数据段，包含 `.sbss` 中未包含的未初始化变量。

C 标准要求未初始化数据段在启动时初始化为 0。要初始化这些段，链接描述文件必须连续地分配这些段，并初始化两个变量——一个表示未初始化数据段的起始地址，一个表示未初始化数据段的结束地址。启动代码会将这两个地址之间的所有数据存储单元清零。这两个变量分别名为 `_bss_begin` 和 `_bss_end`。

图 5-6： 未初始化数据

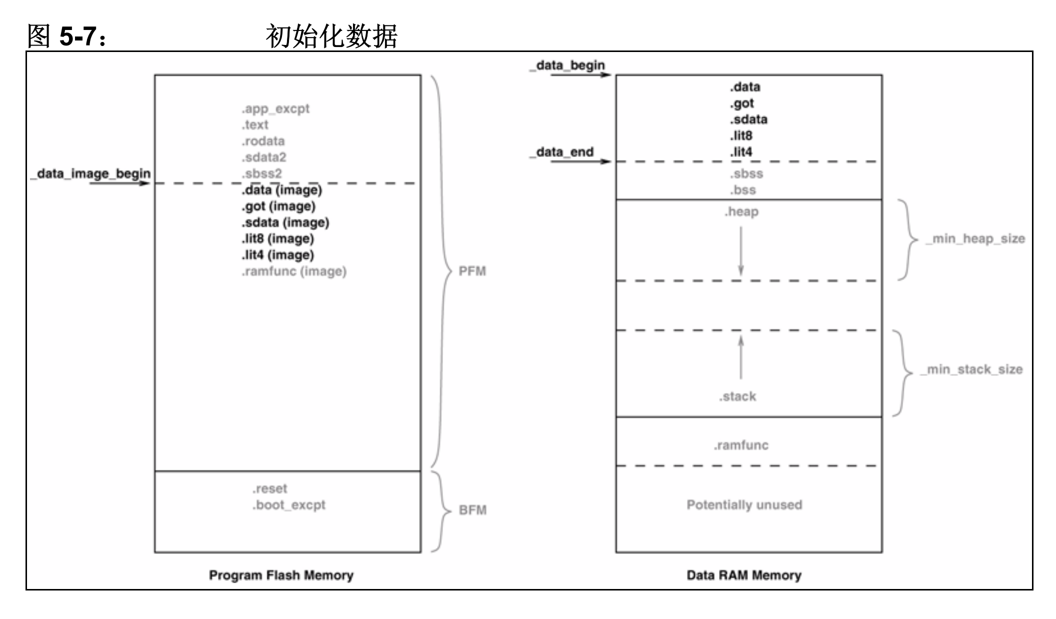


5.7.2.6 将初始化数据从程序闪存复制到数据存储器

类似于未初始化数据段，存在 4 个初始化数据段：`.sdata`、 `.data`、 `.lit4` 和 `.lit8`。 `.sdata` 段是一个数据段，包含长度小于等于 *n* 字节的初始化变量，其中的 *n* 由 -G*n* 命令行选项决定。`.data` 段是一个数据段，包含 `.sdata` 中未包含的初始化变量。 `.lit4` 和 `.lit8` 段包含汇编器决定存储在存储器中而不是存储在指令流中的常量 （通常是浮点数）。

在启动时，程序闪存中存在初始化数据的副本。该数据必须复制到数据存储器中。为了方便该操作，链接描述文件必须初始化 3 个变量——一个表示初始化数据段在程序闪存中镜像的起始地址，一个表示段在数据存储器中的起始地址，一个表示段在数据存储器中的结束地址。启动代码会使用这些变量将所有数据存储单元从程序闪存复制到数据存储器。这些变量分别名为 `_data_image_begin`、 `_data_begin` 和 `_data_end`。

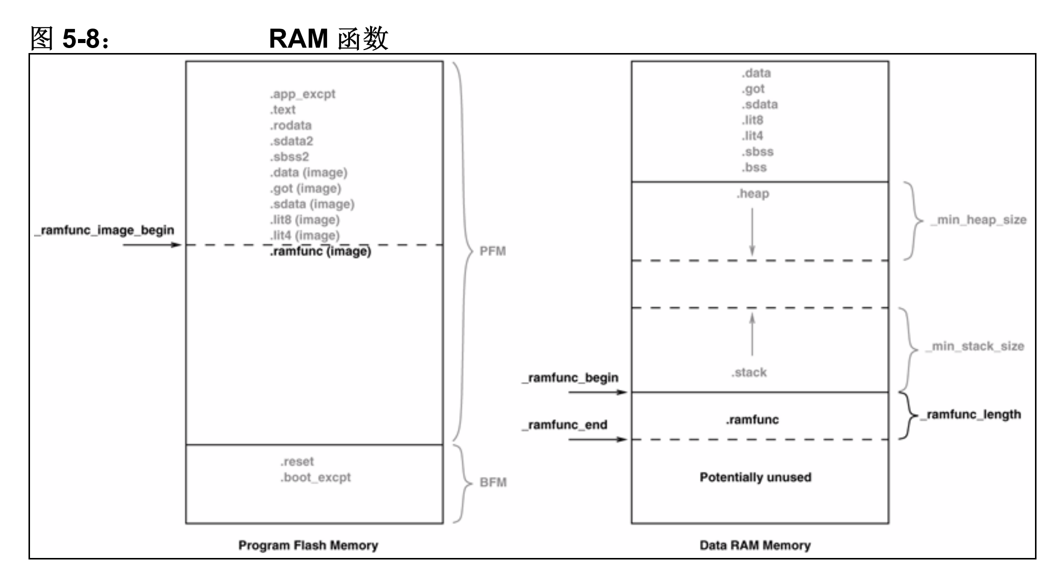
图 **5-7**： 初始化数据



5.7.2.7 将 RAM 函数从程序闪存复制到数据存储器

RAM 函数类似于初始化数据，只是存在于程序闪存中的数据代表的是函数而不是一些符号的初始值。与初始化数据从程序闪存复制到数据存储器的方式类似，链接描述文件必须初始化 3 个变量——一个表示 RAM 函数在程序闪存中镜像的起始地址，一个表示段在数据存储器中的起始地址，一个表示段在数据存储器中的结束地址。启动代码会使用这些变量将存储单元从程序闪存复制到数据存储器。这些变量分别名为 `_ramfunc_image_begin`、 `_ramfunc_begin` 和 `_ramfunc_end`。

图 **5-8**： **RAM** 函数



5.7.2.8 初始化总线矩阵寄存器

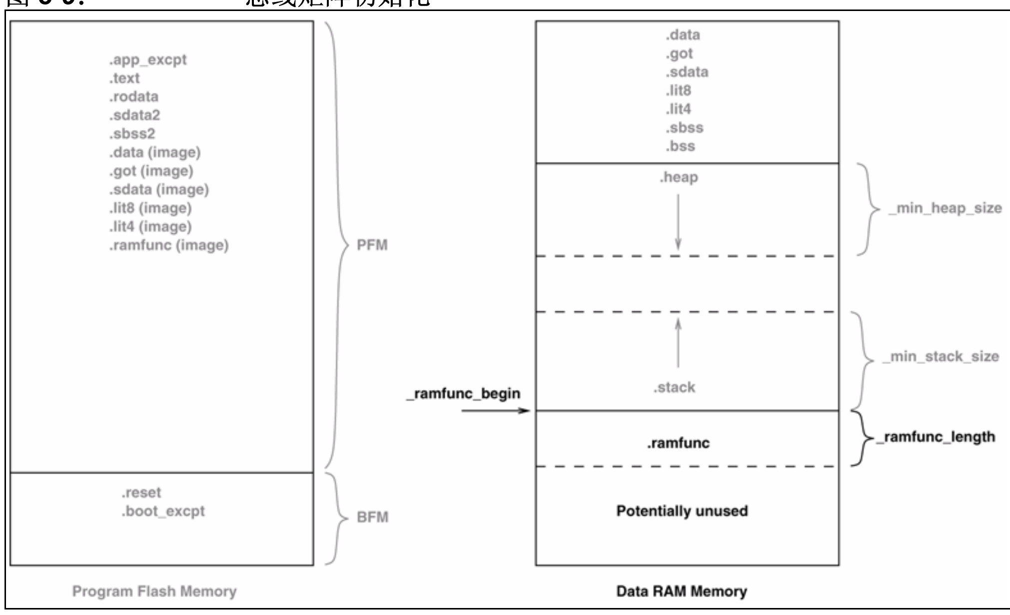
如果存在任何 RAM 函数，那么启动代码应初始化总线矩阵寄存器 （BMXDKPBA、BMXDUDBA 和 BMXDUPBA），否则不应修改这些寄存器。为了确定应用程序中是否存在任何 RAM 函数，链接描述文件提供一个变量，该变量包含了 `.ramfunc` 段[1] 的长度。该变量名为 `_ramfunc_length`。此外，链接描述文件还提供了 3 个变量，这些变量包含总线矩阵寄存器的地址。这些变量名为 `_bmxdkpba_address`、`_bmxdudba_address` 和 `_bmxdupba_address`。以下算式用于计算这些地址：

1. 所有带有属性 `__ramfunc__` 或 `__longramfunc__` 的函数都放入 `.ramfunc` 段。


```
_bmxdkpba_address = _ramfunc_begin -
                    ORIGIN(${DATA_MEMORY_LOCATION}) ;
_bmxdudba_address = LENGTH(${DATA_MEMORY_LOCATION}) ;
_bmxdupba_address = LENGTH(${DATA_MEMORY_LOCATION}) ;
```

链接描述文件会确保 RAM 函数与 2K 边界对齐，这是 BMXDKPBA 寄存器所要求的。

图 5-9： 总线矩阵初始化



5.7.2.9 初始化 CP0 寄存器

CP0 寄存器按以下顺序进行初始化：

1. Count 寄存器
2. Compare 寄存器
3. EBase 寄存器
4. IntCtl 寄存器
5. Cause 寄存器
6. Status 寄存器

5.7.2.9.1 硬件使能寄存器 （HWREna——CP0 寄存器 7，选择 0）

该寄存器包含一个屏蔽位，决定哪些硬件寄存器可通过 RDHWR 指令进行访问。具有特权的软件可以决定其中哪些硬件寄存器可由 RDHWR 指令访问。这么做时，可以通过以下方式将寄存器虚拟化：处理保留指令异常、解释指令并返回虚拟化值。例如，如果不希望直接访问 Count 寄存器，可以单独禁止对该寄存器的访问，返回值可以由操作系统虚拟化。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.2 错误虚拟地址寄存器 （BadVAddr——CP0 寄存器 8，选择 0）

该寄存器是只读寄存器，它捕获导致 “地址错误”异常 （AdEL 或 AdES）的最近的虚拟地址。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.3 Count 寄存器 （Count——CP0 寄存器 9，选择 0）

该寄存器用作一个定时器，无论是否有指令执行、退出，还是指令流水线在向前进行，它都以固定速率递增。如果 Cause 寄存器中的 DC 位为 0，那么计数器每隔一个时钟递增一次。可设置 Count 寄存器的值用于实现功能或进行诊断，包括在复位时使用或用于同步处理器。通过写 Debug 寄存器中的 $Count_{DM}$ 位，可以控制当处理器处于调试模式时， Count 寄存器是否继续递增。

PIC32MX 启动代码会将该寄存器清零。

5.7.2.9.4 Compare 寄存器 （Compare——CP0 寄存器 11，选择 0）

该寄存器与 Count 寄存器联合使用，用于实现定时器和定时器中断功能。定时器中断是内核的输出。 Compare 寄存器会维持一个稳定的值，不会自己更改它。当 Count 寄存器的值等于 Compare 寄存器的值时， SI_TimerInt 引脚输出有效。该引脚保持有效，直到向 Compare 寄存器写入值为止。 SI_TimerInt 引脚的信号可以反馈到内核的一个中断引脚，以产生中断。用于诊断时， Compare 寄存器是读 / 写寄存器。但在正常使用时， Compare 寄存器是只写的。向 Compare 寄存器写入值会产生一个副作用，即清除定时器中断。

在 PIC32MX 启动代码中，会将该寄存器设为 0xFFFFFFFF。

5.7.2.9.5 Status 寄存器 （Status——CP0 寄存器 12，选择 0）

该寄存器是读 / 写寄存器，其中包含处理器的工作模式、中断允许和诊断状态。该寄存器的一些位域联合决定处理器的工作模式。

PIC32MX 启动代码会初始化以下设置
（0b000000000x0xx0?00000000000000000）：

- 在用户模式下不允许访问协处理器 0 （CU0 = 0）
- 用户模式使用已配置的尾数表示法 （RE = 0）
- 不更改异常向量位置 （BEV = no change）
- 不更改指示复位异常向量进入原因的标志位 （SR、 NMI = no change）
- 如果已实现了 CorExtend 用户定义的指令 （$Config_{UDI}$ == 1），则使能 CorExtend （CEE = 1），否则禁止 CorExtend （CEE = 0）。
- 清除中断屏蔽，以禁止所有待处理的中断请求（IM7..IM2 = 0、IM1..IM0 = 0）
- 中断优先级为 0 （IPL = 0）
- 基本模式为内核模式 （UM = 0）
- 错误级别为一般 （ERL = 0）
- 异常级别为一般 （EXL = 0）
- 禁止中断 （IE = 0）

5.7.2.9.6 中断控制寄存器 （IntCtl——CP0 寄存器 12，选择 1）

该寄存器控制在架构的发行版 2 中新增的扩展中断功能，包括向量化中断和对外部中断控制器的支持。

该寄存器中包含用于中断处理的向量间隔。 PIC32MX 启动代码会使用 _vector_spacing 符号的值来初始化该寄存器的向量间隔部分 （bits 9..5）。所有其他位均置 1。

5.7.2.9.7 影子寄存器控制寄存器 （SRSCtl——CP0 寄存器 12，选择 2）

该寄存器控制处理器中的 GPR 影子寄存器组的操作。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.8 影子寄存器映射寄存器 （SRSMap——CP0 寄存器 12，选择 3）

该寄存器包含 8 个 4 位的位域，提供从向量号到在提供中断处理服务时使用的影子寄存器组号的映射。该寄存器的值不用于非中断异常或非向量化中断 （$Cause_{IV}$ = 0 或 $IntCtl_{VS}$ = 0）。在这种情况下，影子寄存器组号来自 $SRSCtl_{ESS}$。如果 $SRSCtl_{HSS}$ 为 0，那么软件读或写该寄存器的结果**不可预测**。如果向该寄存器的任意位域中写入的值大于 $SRSCtl_{HSS}$ 的值，那么处理器的操作是**未定义的**。 SRSMap 寄存器包含对应于向量号 7..0 的影子寄存器组号。对于多个中断向量，可以设定同一影子寄存器组号，产生从向量到单个影子寄存器组号的多对一映射。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.9 Cause 寄存器 （Cause——CP0 寄存器 13，选择 0）

该寄存器主要描述最近异常的原因。此外，一些位域还控制软件中断请求和分派中断向量。 DC、 IV 和 IP1..IP0 位域除外， Cause 寄存器中的所有其他位域都是只读的。架构的发行版 2 新增了对于外部中断控制器 （External Interrupt Controller，EIC）中断模式的可选支持，在该模式下， IP7..IP2 解释为请求的中断优先级（Requested Interrupt Priority Level， RIPL）。

PIC32MX 启动代码会初始化以下设置：

- 使能 Count 寄存器的计数 （DC = 不变）
- 使用特殊异常向量 （16#200）（IV = 1）
- 禁止软件中断请求 （IP1..IP0 = 0）

5.7.2.9.10 异常程序计数器 （EPC——CP0 寄存器 14，选择 0）

该寄存器是读 / 写寄存器，包含在执行异常服务之后恢复处理的地址。 EPC 寄存器的所有位都是有效位，必须可写。对于同步 （精确）异常， EPC 包含以下值之一：

- 直接导致异常的指令的虚拟地址
- 紧接在转移或跳转指令前面的虚拟地址 （如果导致异常的指令处于转移延时入口点，并且 Cause 寄存器中的 Branch Delay 位置 1）。

在发生新的异常时，如果 Status 寄存器中的 EXL 位置 1，那么处理器不会写 EPC 寄存器；但是，仍然可以通过 MTC0 指令写该寄存器。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.11　处理器标识寄存器（PRid——CP0 寄存器 15，选择 0）

该寄存器是 32 位只读寄存器，包含指示制造商、制造商选项、处理器标识和处理器版本号的信息。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.12　异常基址寄存器（EBase——CP0 寄存器 15，选择 1）

该寄存器是读 / 写寄存器，包含在 $Status_{BEV}$ 等于 0 时使用的异常向量的基地址，以及一个只读的 CPU 编号值，软件可以使用该值来区分多处理器系统中的不同处理器。EBase 寄存器使软件可以识别多处理器系统中的特定处理器，使每个处理器的异常向量可以不同，特别是在由异构处理器组成的系统中。当 $Status_{BEV}$ 为 0 时， EBase 寄存器的 bit 31..12 接上 0 构成异常向量的基址。当 $Status_{BEV}$ 为 1 时，或者对于任何 EJTAG 调试异常，异常向量基地址来自固定的默认值。 EBase 寄存器的 bit 31..12 的复位状态会将异常基址寄存器初始化为 16#80000000，与发行版 1 向后兼容。EBase 寄存器的 bit 31..30 固定为值 2#10，以强制使异常基地址处于 KSEG0 或 KSEG1 未映射的虚拟地址段中。

如果要更改异常基址寄存器的值，那么必须在 $Status_{BEV}$ 等于 1 时进行更改。如果在 $Status_{BEV}$ 为 0 时，使用不同的值写异常基址位域，那么处理器的操作是**未定义的**。

bit 31..30 与异常基址位域结合使用时，可以允许将异常向量的基地址放置在任意的 4K 字节页面边界处。如果使用了向量化中断，那么可以产生大于 4K 字节的向量偏移。这种情况下，异常基址位域的 bit 12 必须为 0。如果软件向异常基址位域的 bit 12 写入 1，并使能使用相对于异常基地址的偏移大于 4K 字节的向量化中断，那么处理器的操作是**未定义的**。

PIC32MX 启动代码会使用 _ebase_address 符号的值来初始化该寄存器。链接描述文件会为 _ebase_address 提供默认值，即 KSEG1 程序存储器的起始地址。用户可以通过向链接器提供命令行选项 --defsym _ebase_address=*A* 来更改该值。

5.7.2.9.13　Config 寄存器（Config——CP0 寄存器 16，选择 0）

该寄存器指定各种配置和功能信息。 Config 寄存器中的大部分位域由硬件在复位异常过程中初始化，或者是常量。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.14　Config1 寄存器（Config1——CP0 寄存器 16，选择 1）

该寄存器是 Config 寄存器的附属寄存器，保存关于内核中提供的功能的其他信息。Config1 寄存器中的所有位域都是只读的。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.15　Config2 寄存器（Config2——CP0 寄存器 16，选择 2）

该寄存器是 Config 寄存器的附属寄存器，保留用于保存其他功能信息。分配 Config2 用于显示 2/3 级高速缓存的配置。这些位域复位为 0，因为内核不支持 L2/L3 高速缓存。 Config2 寄存器中的所有位域都是只读的。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.16 Config3 寄存器 （Config3——CP0 寄存器 16，选择 3）

该寄存器保存其他功能信息。 Config3 寄存器中的所有位域都是只读的。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.17 Debug 寄存器 （Debug——CP0 寄存器 23，选择 0）

该寄存器用于控制调试异常，提供关于调试异常产生原因的信息，以及关于在调试模式下由于一般异常而重新进入调试异常向量时的信息。每次捕获调试异常或在调试模式下捕获一般异常时，都会更新这些只读信息位。在非调试模式下读取时，只有 DM 位和 $EJTAG_{ver}$ 位域是有效的。所有其他位和位域的值是**不可预测的**。如果在非调试模式下写 Debug 寄存器，那么处理器的操作是**未定义的**。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.9.18 跟踪控制寄存器 （TraceControl——CP0 寄存器 23，选择 1）

该寄存器提供控制和状态信息。只有提供 EJTAG 跟踪功能时，才会实现 TraceControl 寄存器。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.10 跟踪控制 2 寄存器 （TraceControl2——CP0 寄存器 23，选择 2）

该寄存器提供其他控制和状态信息。只有提供 EJTAG 跟踪功能时，才会实现 TraceControl2 寄存器。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.10.1 用户跟踪数据寄存器 （UserTraceData——CP0 寄存器 23，选择 3）

写该寄存器时，将会写入一条跟踪记录，并指示是类型 1 还是类型 2 用户格式。该类型基于 TraceControl 寄存器中的 UT 位。不能在连续的周期中写该寄存器。如果在连续周期中写该寄存器，跟踪输出数据是**不可预测的**。只有提供 EJTAG 跟踪功能时，才会实现 UserTraceData 寄存器。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.10.2 TraceBPC 寄存器 （TraceBPC——CP0 寄存器 23，选择 4）

该寄存器用于通过使用 EJTAG 硬件断点控制跟踪的启动和停止。这时，硬件断点将被设为触发源，并且还可选作调试异常断点。只有同时提供硬件断点和 EJTAG 跟踪功能时，才会实现 TraceBPC 寄存器。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.10.3 Debug2 寄存器 （Debug2——CP0 寄存器 23，选择 5）

该寄存器保存更多关于复杂断点异常的信息。只有提供复杂硬件断点时，才会实现 Debug2 寄存器。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.10.4 调试异常程序计数器 （DEPC——CP0 寄存器 24，选择 0）

该寄存器是读 / 写寄存器，包含在执行调试异常或调试模式异常服务之后恢复处理的地址。对于同步 （精确）调试和调试模式异常， DEPC 包含以下值之一：

- 直接导致调试异常的指令的虚拟地址，或者
- 紧接在转移或跳转指令前面的虚拟地址 （如果导致调试异常的指令处于跳转延时时隙中，并且 Debug 寄存器中的调试转移延时 DBD 位置 1 时）。

对于异步调试异常 （调试中断和复杂断点）， DEPC 包含在执行调试处理程序代码之后应继续开始执行的指令的虚拟地址。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.10.5 错误异常程序计数器 （ErrorEPC——CP0 寄存器 30，选择 0）

该寄存器是读 / 写寄存器，类似于 EPC 寄存器，只是它是在发生错误异常时使用。ErrorEPC 寄存器的所有位都是有效位，必须可写。它还用于在复位、软复位和不可屏蔽中断 （NMI）异常时存储程序计数器。 ErrorEPC 寄存器包含在进行错误处理之后，继续开始执行的指令的虚拟地址。该地址可以是：

- 导致异常的指令的虚拟地址，或者
- 紧接在转移或跳转指令前面的虚拟地址 （如果导致错误的指令处于转移延时时隙中）。

不同于 EPC 寄存器， ErrorEPC 寄存器没有相应的转移延时时隙指示。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.10.6 调试异常保存寄存器 （DeSave——CP0 寄存器 31，选择 0）

该寄存器是读 / 写寄存器，用作简单的存储单元。该寄存器供调试异常处理程序保存一个 GPR，该 GPR 会在随后用于将现场的其他信息保存到预先确定的存储区 （例如，保存在 EJTAG 探测器中）。在无法确定是否存在有效的堆栈用于保存现场时，该寄存器使得可以安全地调试异常处理程序和其他类型的代码。

PIC32MX 启动代码中不对该寄存器执行初始化。

5.7.2.11 调用 “自举时”过程

在初始化 CP0 寄存器之后，将会调用一个过程。该过程使用户可以在自举期间 （即，当 Status_{BEV} 置 1 时）和在进入主程序之前执行一些操作。随启动代码一起，提供了该过程 （_on_bootstrap）的弱版本，其内容为空。该过程可以用于执行硬件初始化和 / 或初始化 RTOS 所需的环境。

5.7.2.12 更改异常向量的位置

在调用应用程序主程序之前， Status_{BEV} 会被清零，以将异常向量的位置从自举位置更改为正常位置。

5.7.2.13 调用主程序

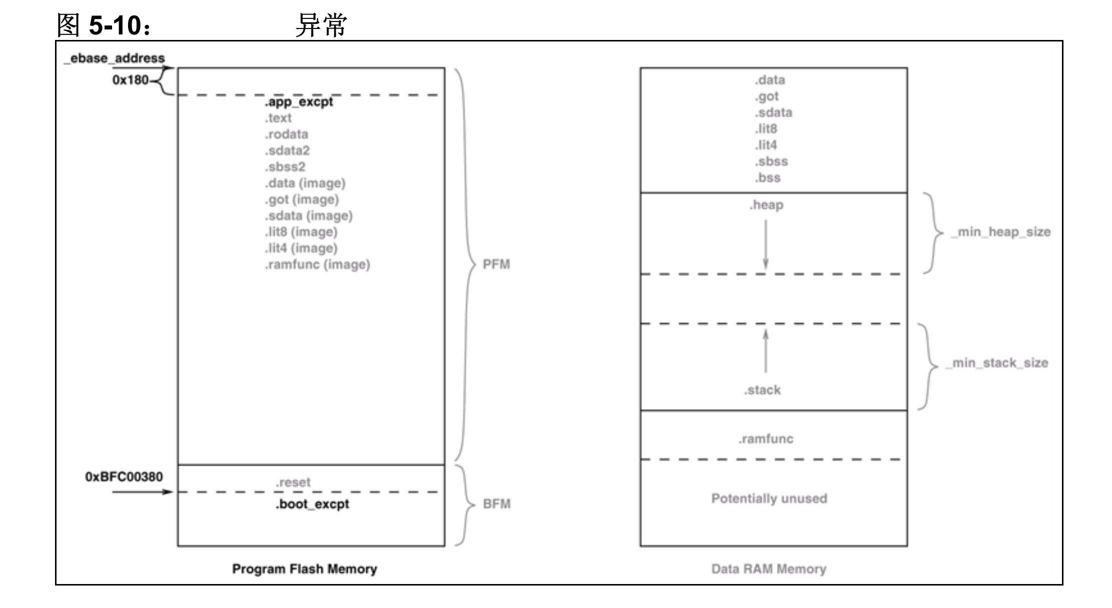
启动代码执行的最后一项操作是调用主程序。如果用户从主程序返回，启动代码将进入无限循环。

5.7.3 异常

此外，还提供了两个弱版本的通用异常处理程序，可以由应用程序覆盖——一个处理 $Status_{BEV}$ 为 1 时的异常（_bootstrap_exception_handler），一个处理 $Status_{BEV}$ 为 0 时的异常（_general_exception_handler）。随启动代码提供的弱版本的复位异常处理程序和一般异常处理程序都进入一个无限循环。启动代码会进行以下安排：目标为复位异常处理程序的跳转定位到 0xBFC00380，目标为一般异常处理程序的跳转定位到 EBASE + 0x180。

两个处理程序都必须加上属性 nomips16，[例如， __attribute__ ((nomips16))]，因为启动代码会跳转到这些函数。

图 5-10： 异常



5.7.4 启动代码和 C 函数库需要的符号

本节详细说明启动代码和 C 函数库所需的符号。当前，默认的链接描述文件定义了以下符号。如果应用程序提供了一个定制的链接描述文件，那么用户必须确保提供以下所有符号，以便启动代码和 C 函数库发挥作用：

符号名称	说明
`_bmxdkpba_address`	在 `_ramfunc_length` 大于 0 时，要放入 BMXDKPBA 寄存器的地址。
`_bmxdudba_address`	在 `_ramfunc_length` 大于 0 时，要放入 BMXDUDBA 寄存器的地址。
`_bmxdupba_address`	在 `_ramfunc_length` 大于 0 时，要放入 BMXDUPBA 寄存器的地址。
`_bss_begin`	未初始化数据的起始地址。未初始化数据包括 `.sbss` 和 `.bss`。
`_bss_end`	未初始化数据的结束地址。未初始化数据包括 `.sbss` 和 `.bss`。
`_data_begin`	初始化数据的起始地址。初始化数据包括 `.data`、`.got`、`.sdata`、`.lit8` 和 `.lit4`。
`_data_end`	初始化数据的结束地址。初始化数据包括 `.data`、`.got`、`.sdata`、`.lit8` 和 `.lit4`。
`_data_image_begin`	初始化数据的镜像在程序存储器中的起始地址。初始化数据包括 `.data`、`.got`、`.sdata`、`.lit8` 和 `.lit4`。
`_ebase_address`	EBASE 的地址。
`_end`	数据分配的结束地址。应与 `_heap` 相同。
`_gp`	指向小变量段的 “中间”。依照约定，它位于从用于小长度变量的第一个单元偏移 0x8000 字节的位置。
`_heap`	DRM 中的堆的起始地址。
`_ramfunc_begin`	RAM 函数的起始地址。它应位于 2K 边界处，因为它用于初始化 BMXDKPBA 寄存器。
`_ramfunc_end`	RAM 函数的结束地址。
`_ramfunc_image_begin`	RAM 函数的镜像在程序存储器中的起始地址。
`_ramfunc_length`	`.ramfunc` 段的长度。
`_stack`	DRM 中的堆栈的起始地址。请记住，堆栈是从数据存储区的底部开始增长，所以该符号应指向为堆栈分配的段的底部。
`_vector_spacing`	`IntCtl` 寄存器中的向量间隔位域的初始化值。

5.8 默认链接描述文件的内容

默认链接描述文件包含以下类别的信息：

- 输出格式和入口点
- 最小堆栈和堆大小的默认值
- 处理器定义包含文件

- 包含特定于处理器的目标文件
- 异常向量基地址和向量间隔符号
- 存储器地址赋值
- 存储区域
- 配置字输入 / 输出区域映射

• 输入 / 输出区域映射

注：	在链接描述文件中指定的所有地址应指定为虚拟地址而不是物理地址。

5.8.1 输出格式和入口点

默认链接描述文件的前几行定义应用程序的输出格式和入口点。在 C:\program files\...\MPLAB C32\pic32mx\lib\ldscripts 中提供了默认链接描述文件的副本。

```
OUTPUT_FORMAT("elf32-tradlittlemips")
OUTPUT_ARCH(pic32mx)
ENTRY(_reset)
```

`OUTPUT_FORMAT` 行选择输出文件的目标文件格式。由 MPLAB C32 语言工具生成的输出目标文件格式是传统的小尾数、 MIPS 或 ELF32 格式。

`OUTPUT_ARCH` 行选择输出文件的特定机器架构。由 MPLAB C32 语言工具生成的输出文件包含有指明文件是针对 PIC32MX 架构而生成的信息。

`ENTRY` 行选择应用程序的入口点。这是指明要执行的第一条指令的单元位置的符号。MPLAB C32 语言工具在由 `_reset` 标号指示的指令处开始执行。

5.8.2 最小堆栈和堆大小的默认值

默认链接描述文件的下一个部分提供最小堆栈和堆大小的默认值。

```
/*
 * Provide for a minimum stack and heap size
 * - _min_stack_size - represents the minimum space that must
 *                     be made available for the stack.  Can
 *                     be overridden from the command line
 *                     using the linker's --defsym option.
 * - _min_heap_size  - represents the minimum space that must
 *                     be made available for the heap.  Can
 *                     be overridden from the command line
 *                     using the linker's --defsym option.
 */
EXTERN (_min_stack_size _min_heap_size)
PROVIDE(_min_stack_size = 0x400) ;
PROVIDE(_min_heap_size = 0) ;
```

`EXTERN` 行确保链接描述文件的其余部分可以访问 `_min_stack_size` 和 `_min_heap_size` 的默认值（假定用户未使用链接器的 `--defsym` 命令行选项覆盖这些值）。

两个 `PROVIDE` 行确保为 `_min_stack_size` 和 `_min_heap_size` 都提供了默认值。最小堆栈大小的默认值为 `1024` 字节（`0x400`）。最小堆大小的默认值为 `0` 字节。

5.8.3 处理器定义包含文件

默认链接描述文件的下一行引入特定于处理器的信息。

```
INCLUDE procdefs.ld
```

文件 procdefs.ld 在此处被包含到链接描述文件中。在当前目录和在使用 -L 命令行选项指定的任意目录中搜索该文件。编译器 shell 确保基于使用 -mprocessor 命令行选项选择的处理器，使用 -L 命令行选项向链接器传递正确的目录。

处理器定义链接描述文件包含以下一些信息：

- 包含特定于处理器的目标文件
- 异常向量基地址和向量间隔符号
- 存储器地址赋值
- 存储区域
- 配置字输入 / 输出区域映射

5.8.3.1 包含特定于处理器的目标文件

处理器定义链接描述文件的该部分确保在链接中包含特定于处理器的目标文件。

```
/*************************************************************
 * Processor-specific object file.  Contains SFR definitions.
 *************************************************************/
INPUT("processor.o")
```

INPUT 行指定 processor.o 应包含到链接中，就如同是在命令行中指定了该文件。链接器会尝试在当前目录中查找该文件。如果未找到，那么链接器会搜索所有库搜索路径 （即，使用 -L 命令行选项指定的路径）。

5.8.3.2 异常向量基地址和向量间隔符号

处理器定义链接描述文件的该部分定义异常向量基地址和向量间隔的值。

```
/*************************************************************
 * For interrupt vector handling
 *************************************************************/
_vector_spacing= 0x00000001;
_ebase_address= 0x9FC01000;
```

第一行为 _vector_spacing 定义值 1。可供异常使用的存储器仅支持值为 1 的向量间隔。第二行定义异常向量基地址的位置 （EBASE）。该地址位于 KSEG0 引导段中。

5.8.3.3 存储器地址赋值

处理器定义链接描述文件的该部分提供关于默认链接描述文件所需的一些特定存储器地址的信息。

```
/*************************************************************
 * Memory Address Equates
 *************************************************************/
_RESET_ADDR= 0xBFC00000;
_BEV_EXCPT_ADDR= 0xBFC00380;
_DBG_EXCPT_ADDR= 0xBFC00480;
_DBG_CODE_ADDR= 0xBFC02000;
_GEN_EXCPT_ADDR= _ebase_address + 0x180;
```

_RESET_ADDR 定义处理器的复位地址。这是在内核模式下， IFM 引导段的虚拟起始地址。

_BEV_EXCPT_ADDR 定义在遇到异常且 $Status_{BEV}$ = 1 时处理器跳转到的地址。

_DBG_EXCPT_ADDR 定义在遇到调试异常时处理器跳转到的地址。

_DBG_CODE_ADDR 定义调试执行程序的起始地址。

_GEN_EXCPT_ADDR 定义在遇到异常且 $Status_{BEV}$ = 0 时处理器跳转到的地址。

5.8.3.4 存储段

处理器定义链接描述文件的该部分提供关于器件上可用的存储段的信息。

```
/*************************************************************
 * Memory Regions
 *
 * Memory regions without attributes cannot be used for
 * orphaned sections.  Only sections specifically assigned to
 * these regions can be allocated into these regions.
 *************************************************************/
MEMORY
{
  kseg0_program_mem (rx) : ORIGIN = 0x9D000000, LENGTH = 0x8000
  kseg0_boot_mem         : ORIGIN = 0x9FC00490, LENGTH = 0x970
  exception_mem          : ORIGIN = 0x9FC01000, LENGTH = 0x1000
  kseg1_boot_mem         : ORIGIN = 0xBFC00000, LENGTH = 0x490
  debug_exec_mem         : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
  config3                : ORIGIN = 0xBFC02FF0, LENGTH = 0x4
  config2                : ORIGIN = 0xBFC02FF4, LENGTH = 0x4
  config1                : ORIGIN = 0xBFC02FF8, LENGTH = 0x4
  config0                : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
  kseg1_data_mem (w!x)   : ORIGIN = 0xA0000000, LENGTH = 0x2000
  sfrs                   : ORIGIN = 0xBF800000, LENGTH = 0x10000
}
```

11 个存储段使用相关的起始地址和长度进行定义：

1. 用于应用程序代码的程序存储区 （kseg0_program_mem）
2. 引导存储区 （kseg0_boot_mem 和 kseg1_boot_mem）
3. 异常存储区 （exception_mem）
4. 调试执行程序存储区 （debug_exec_mem）
5. 配置存储区 （config3、 config2、 config1 和 config0）
6. 数据存储区 （kseg1_data_mem）
7. SFR 存储区 （sfrs）

默认链接描述文件使用这些名称来将各个代码段定位到正确的存储区中。非标准代码段会成为孤立的代码段。存储区的属性用于定位这些孤立代码段。属性 （rx）指定只读代码段或可执行代码段可以定位到程序存储区中。类似地，属性 （w!x）指定非只读和不可执行代码段可以定位到数据存储区中。因为未为引导存储区、配置存储区或 SFR 存储区指定任何属性，所以只有指定的代码段可以定位到这些存储区中 （即，孤立代码段不能定位到引导存储区、异常存储区、配置存储区、调试执行程序存储区或 SFR 存储区）。

5.8.3.5 配置字输入 / 输出段映射

处理器定义链接描述文件的最后一个部分是配置字的输入 / 输出段映射。该部分是对默认链接描述文件中输入 / 输出段映射的补充（见第 **5.8.4 节 "输入 / 输出区域映射"**）。它定义如何将配置字的输入段映射到配置字的输出段。请注意，输入段是在源代码中定义的应用程序的组成部分，而输出段则由链接器创建。通常，几个输入段可以组合为单个输出段。所有输出段都在链接描述文件的 `SECTIONS` 命令中指定。

对于特定处理器中存在的每个配置字，都存在一个名为 `.config_`*`address`* 的独特输出段，其中的 address 是配置字在存储器中的存储地址。这些段中的每一个都包含由 `#pragma config` 伪指令（见第 **4.8.1 节 "`#pragma config`"**）为该配置字创建的数据。每个段都被分配给它们各自的存储区（`config`*`n`*）。

```
SECTIONS
{
  .config_BFC02FF0 : {
    *(.config_BFC02FF0)
  } > config3
  .config_BFC02FF4 : {
    *(.config_BFC02FF4)
  } > config2
  .config_BFC02FF8 : {
    *(.config_BFC02FF8)
  } > config1
  .config_BFC02FFC : {
    *(.config_BFC02FFC)
  } > config0
}
```

5.8.4 输入 / 输出段映射

默认链接描述文件中的最后一个部分是输入 / 输出段映射。段映射是链接描述文件的核心。它定义如何将输入段映射到输出段。请注意，输入段是在源代码中定义的应用程序的组成部分，而输出段则由链接器创建。通常，几个输入段可以组合为单个输出段。所有输出段都在链接描述文件的 `SECTIONS` 命令中指定。

链接器可能会创建以下输出段：

- .reset 段
- .bev_excpt 段
- .DBG_excpt 段
- .dbg_code 段
- .app_excpt 段
- .vector_0 ...vector_63 段
- .startup 段
- .text 段
- .rodata 段

- .sdata2 段
- .sbss2 段
- .dbg_data 段
- .data 段
- .got 段
- .sdata 段
- .lit8 段
- .lit4 段
- .sbss 段
- .bss 段
- .heap 段
- .stack 段
- .ramfunc 段
- 堆栈位置
- 调试区域

5.8.4.1 .RESET 段

该段包含在处理器进行复位时执行的代码。该段定位到在处理器定义链接描述文件中指定的复位地址 （_RESET_ADDR）处，并分配给引导存储区（kseg1_boot_mem）。

```
.reset _RESET_ADDR :
{
  *(.reset)
} > kseg1_boot_mem
```

5.8.4.2 .BEV_EXCPT 段

该段包含在 $Status_{BEV}$ = 1 时发生的异常的处理程序。该段定位到在处理器定义链接描述文件中指定的 BEV 异常地址 （_BEV_EXCPT_ADDR）处，并分配给引导存储区（kseg1_boot_mem）。

```
.bev_excpt _BEV_EXCPT_ADDR :
{
  *(.bev_handler)
} > kseg1_boot_mem
```

5.8.4.3 .DBG_EXCPT 段

该段用于为调试异常向量保留空间。只有定义了符号 _DEBUGGER 时，才会分配该段。（如果对 shell 指定了 -mdebugger 命令行选项，就定义了该符号。）该段定位到在处理器定义链接描述文件中指定的调试异常地址 （_DBG_EXCPT_ADDR）处，并分配给引导存储区 （kseg1_boot_mem）。该段标记为 NOLOAD，因为它只是用于确保应用程序代码不会放置在为调试执行程序保留的位置。

```
.dbg_excpt _DBG_EXCPT_ADDR (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0x8 : 0x0);
} > kseg1_boot_mem
```

5.8.4.4 .DBG_CODE 段

该段用于为调试异常处理程序保留空间。只有定义了符号 _DEBUGGER 时，才会分配该段。（如果对 shell 指定了 -mdebugger 命令行选项，就定义了该符号。）该段定位到在处理器定义链接描述文件中指定的调试代码地址 （_DBG_CODE_ADDR）处，并分配给调试执行程序存储区 （debug_exec_mem）。该段标记为 NOLOAD，因为它只是用于确保应用程序代码不会放置在为调试执行程序保留的位置。

```
.dbg_code _DBG_CODE_ADDR (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0xFF0 : 0x0);
} > debug_exec_mem
```

5.8.4.5 .APP_EXCPT 段

该段包含在 Status_{BEV} = 0 时发生的异常的处理程序。该段定位到在处理器定义链接描述文件中指定的一般异常地址 （_GEN_EXCPT_ADDR）处，并分配给异常存储区（exception_mem）。

```
.app_excpt _GEN_EXCPT_ADDR :
{
  *(.gen_handler)
} > exception_mem
```

5.8.4.6 .VECTOR_0 ...VECTOR_63 段

这些段包含每个中断向量的处理程序。这些段使用以下公式定位到正确的向量地址处：

```
_ebase_address + 0x200 + (_vector_spacing << 5) * n
```

其中， n 是各个向量号。

每个段后都跟随一个断言，确保定位到向量处的代码不超出所指定的向量间隔。

```
.vector_n _ebase_address + 0x200 + (_vector_spacing << 5) * n :
  {
    *(.vector_n)
  } > exception_mem
ASSERT (SIZEOF(.vector_n) < (_vector_spacing << 5), "function at
exception vector n too large")
```

5.8.4.7 .STARTUP 段

该段包含 C 启动代码，它被分配到 KSEG0 引导存储区 （kseg0_boot_mem）。

```
.startup ORIGIN(kseg0_boot_mem) :
  {
    *(.startup)
  } > kseg0_boot_mem
```

5.8.4.8 .TEXT 段

该段从应用程序的所有输入文件中收集可执行代码。该段分配给程序存储区（kseg0_program_mem），并填入值 NOP (0)。已定义了一些符号，用于表示该段的开始 （_text_begin）和结束 （_text_end）地址。

```
.text ORIGIN(kseg0_program_mem) :
{
  _text_begin = . ;
```

```
  *(.text .stub .text.* .gnu.linkonce.t.*)
  KEEP (*(.text.*personality*))
  *(.gnu.warning)
  *(.mips16.fn.*)
  *(.mips16.call.*)
  _text_end = . ;
} > kseg0_program_mem =0
```

5.8.4.9 .RODATA 段

该段从应用程序的所有输入文件中收集只读段。该段分配给程序存储区（kseg0_program_mem）。

```
.rodata   :
{
  *(.rodata .rodata.* .gnu.linkonce.r.*)
  *(.rodata1)
} > kseg0_program_mem
```

5.8.4.10 .SDATA2 段

该段从应用程序的所有输入文件中收集小长度的初始化全局和静态数据常量。因为数据的常量特性，该段也是只读段。该段分配给程序存储区（kseg0_program_mem）。

```
/*
 * Small initialized constant global and static data can be
 * placed in the .sdata2 section.  This is different from
 * .sdata, which contains small initialized non-constant
 * global and static data.
 */
.sdata2   :
{
  *(.sdata2 .sdata2.* .gnu.linkonce.s2.*)
} > kseg0_program_mem
```

5.8.4.11 .SBSS2 段

该段从应用程序的所有输入文件中收集小长度的未初始化全局和静态数据常量。因为数据的常量特性，该段也是只读段。该段分配给程序存储区（kseg0_program_mem）。

```
/*
 * Uninitialized constant global and static data (i.e.,
 * variables which will always be zero).  Again, this is
 * different from .sbss, which contains small non-initialized,
 * non-constant global and static data.
 */
.sbss2   :
{
  *(.sbss2 .sbss2.* .gnu.linkonce.sb2.*)
} > kseg0_program_mem
```

5.8.4.12 .DBG_DATA 段

该段用于为调试异常处理程序所需的数据保留空间。只有定义了符号 _DEBUGGER 时，才会分配该段。（如果对 shell 指定了 -mdebugger 命令行选项，就定义了该符号。）该段分配给数据存储区 （kseg1_data_mem）。该段标记为 NOLOAD，因为它只是用于确保应用程序数据不会存放在为调试执行程序保留的位置。

```
.dbg_data (NOLOAD) :
{
  . += (DEFINED (_DEBUGGER) ? 0x200 : 0x0);
} > kseg1_data_mem
```

5.8.4.13 .DATA 段

该段从应用程序的所有输入文件中收集初始化数据。该段分配给数据存储区（kseg1_data_mem），装入地址定位到程序存储区 （kseg0_program_mem）。已定义了一些符号，用于表示该段的虚拟起始 （_data_begin）和结束（_data_end）地址，以及数据在程序存储器中的物理起始地址（_data_image_begin）。

```
.data   :
{
  _data_begin = . ;
  *(.data .data.* .gnu.linkonce.d.*)
  KEEP (*(.gnu.linkonce.d.*personality*))
  *(.data1)
} > kseg1_data_mem AT> kseg0_program_mem
_data_image_begin = LOADADDR(.data) ;
```

5.8.4.14 .GOT 段

该段从应用程序的所有输入文件中收集全局偏移表。该段分配给数据存储区（kseg1_data_mem），装入地址定位到程序存储区 （kseg0_program_mem）。已定义了一个符号，用于表示全局指针的单元位置 （_gp）。

```
  _gp = ALIGN(16) + 0x7FF0 ;
  .got   :
  {
     *(.got.plt) *(.got)
  } > kseg1_data_mem AT> kseg0_program_mem
```

5.8.4.15 .SDATA 段

该段从应用程序的所有输入文件中收集小长度的初始化数据。该段分配给数据存储区（kseg1_data_mem），装入地址定位到程序存储区 （kseg0_program_mem）。已定义了一些符号，用于表示该段的虚拟起始 （_sdata_begin）和结束（_sdata_end）地址。

```
/*
 * We want the small data sections together, so
 * single-instruction offsets can access them all, and
 * initialized data all before uninitialized, so
 * we can shorten the on-disk segment size.
 */
.sdata   :
{
  _sdata_begin = . ;
  *(.sdata .sdata.* .gnu.linkonce.s.*)
  _sdata_end = . ;
} > kseg1_data_mem AT> kseg0_program_mem
```

5.8.4.16 .LIT8 段

该段从应用程序的所有输入文件中，收集汇编器决定存储在存储器而不是指令流中的 8 字节常量 （通常是浮点数）。该段分配给数据存储区 （kseg1_data_mem），装入地址定位到程序存储区 （kseg0_program_mem）。

```
.lit8          :
{
  *(.lit8)
} > kseg1_data_mem AT> kseg0_program_mem
```

5.8.4.17 .LIT4 段

该段从应用程序的所有输入文件中，收集汇编器决定存储在存储器而不是指令流中的 4 字节常量 （通常是浮点数）。该段分配给数据存储区 （kseg1_data_mem），装入地址定位到程序存储区 （kseg0_program_mem）。已定义了一个符号，用于表示初始化数据的虚拟结束地址 （_data_end）。

```
  .lit4          :
  {
    *(.lit4)
  } > kseg1_data_mem AT> kseg0_program_mem
  _data_end = . ;
```

5.8.4.18 .SBSS 段

该段从应用程序的所有输入文件中收集小长度的未初始化数据。该段分配给数据存储区 （kseg1_data_mem）。已定义了一个符号，用于表示未初始化数据的虚拟起始地址 （_bss_begin）。另外还定义了一些符号，用于表示该段的虚拟起始 （_sbss_begin）和结束 （_sbss_end）地址。

```
  _bss_begin = . ;
  .sbss   :
  {
    _sbss_begin = . ;
    *(.dynsbss)
    *(.sbss .sbss.* .gnu.linkonce.sb.*)
    *(.scommon)
    _sbss_end = . ;
  } > kseg1_data_mem
```

5.8.4.19 .BSS 段

该段从应用程序的所有输入文件中收集未初始化数据。该段分配给数据存储区 （kseg1_data_mem）。已定义了一个符号，用于表示未初始化数据的虚拟结束地址 （_bss_end）。另外还定义了一个符号，用于表示数据存储器的虚拟结束地址 （_end）。

```
.bss   :
{
  *(.dynbss)
  *(.bss .bss.* .gnu.linkonce.b.*)
  *(COMMON)
  /*
   * Align here to ensure that the .bss section occupies
   * space up to _end.  Align after .bss to ensure correct
   * alignment even if the .bss section disappears because
   * there are no input sections.
```



```
    */
    . = ALIGN(32 / 8) ;
  } > kseg1_data_mem
  . = ALIGN(32 / 8) ;
  _end = . ;
  _bss_end = . ;
```

5.8.4.20 .HEAP 段

该段用于为堆保留空间，进行动态存储器分配需要使用堆。已定义了一个符号，用于表示堆的虚拟地址 （_heap）。为堆保留的最小空间大小由符号 _min_heap_size 决定。

```
/* Heap allocating takes a chunk of memory following BSS */
.heap ALIGN(4) :
{
  _heap = . ;
  . += _min_heap_size ;
} > kseg1_data_mem
```

5.8.4.21 .STACK 段

该段用于为堆栈保留空间。为堆栈保留的最小空间大小由符号 _min_stack_size 决定。

```
/* Stack allocation follows the heap */
.stack ALIGN(4) :
{
  . += _min_stack_size ;
} > kseg1_data_mem
```

5.8.4.22 .RAMFUNC 段

该段从应用程序的所有输入文件中收集 RAM 函数。该段分配给数据存储区（kseg1_data_mem），装入地址定位到程序存储区 （kseg0_program_mem）。已定义了一些符号，用于表示该段的虚拟起始 （_ramfunc_begin）和结束（_ramfunc_end）地址，以及 RAM 函数在程序存储器中的物理起始地址（_ramfunc_image_begin）和 RAM 函数的长度 （_ramfunc_length）。此外，还会计算总线矩阵寄存器的地址（_bmxdkpba_address、_bmxdudba_address 和 _bmxdupba_address）。

```
/*
 * RAM functions go at the end of our stack and heap allocation.
 * Alignment of 2K required by the boundary register (BMXDKPBA).
 */
.ramfunc ALIGN(2K) :
{
  _ramfunc_begin = . ;
  *(.ramfunc  .ramfunc.*)
  . = ALIGN(4) ;
  _ramfunc_end = . ;
} > kseg1_data_mem AT> kseg0_program_mem
_ramfunc_image_begin = LOADADDR(.ramfunc) ;
_ramfunc_length = SIZEOF(.ramfunc) ;
_bmxdkpba_address = _ramfunc_begin - ORIGIN(kseg1_data_mem) ;
_bmxdudba_address = LENGTH(kseg1_data_mem);
_bmxdupba_address = LENGTH(kseg1_data_mem);
```

5.8.4.23 堆栈位置

已定义了一个符号，用于表示堆栈指针的位置（`_stack`）。该位置取决于应用程序中是否存在 RAM 函数。如果存在 RAM 函数，那么堆栈指针的位置应当包括堆栈段和 `.ramfunc` 段开始位置减小一个字之间的间隙（由于 `.ramfunc` 段对齐而导致）。如果不存在 RAM 函数，那么堆栈指针的位置应为 KSEG1 数据存储区的结束位置。

```
/*
 * The actual top of stack should include the gap between
 * the stack section and the beginning of the .ramfunc
 * section caused by the alignment of the .ramfunc section
 * minus 1 word.  If RAM functions do not exist, then the top
 * of the stack should point to the end of the kseg1 data
 * memory.
 */
_stack = (_ramfunc_length > 0)
       ? _ramfunc_begin - 4
       : ORIGIN(kseg1_data_mem) + LENGTH(kseg1_data_mem) ;
ASSERT((_min_stack_size + _min_heap_size) <= (_stack - _heap),
  "Not enough space to allocate both stack and heap.  Reduce heap
and/or stack size.")
```

5.8.4.24 调试段

调试段包含调试信息。它们不被装入到闪存程序存储器中。

```
/* Stabs debugging sections.  */
.stab          0 : { *(.stab) }
.stabstr       0 : { *(.stabstr) }
.stab.excl     0 : { *(.stab.excl) }
.stab.exclstr  0 : { *(.stab.exclstr) }
.stab.index    0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment       0 : { *(.comment) }
/* DWARF debug sections.
   Symbols in the DWARF debugging sections are relative
   to the beginning of the section so we begin them at 0.  */
/* DWARF 1 */
.debug          0 : { *(.debug) }
.line           0 : { *(.line) }
/* GNU DWARF 1 extensions */
.debug_srcinfo  0 : { *(.debug_srcinfo) }
.debug_sfnames  0 : { *(.debug_sfnames) }
/* DWARF 1.1 and DWARF 2 */
.debug_aranges  0 : { *(.debug_aranges) }
.debug_pubnames 0 : { *(.debug_pubnames) }


/* DWARF 2 */
.debug_info     0 : { *(.debug_info .gnu.linkonce.wi.*) }
.debug_abbrev   0 : { *(.debug_abbrev) }
.debug_line     0 : { *(.debug_line) }
.debug_frame    0 : { *(.debug_frame) }
.debug_str      0 : { *(.debug_str) }
.debug_loc      0 : { *(.debug_loc) }
.debug_macinfo  0 : { *(.debug_macinfo) }
/* SGI/MIPS DWARF 2 extensions */
.debug_weaknames 0 : { *(.debug_weaknames) }
.debug_funcnames 0 : { *(.debug_funcnames) }
```

```
.debug_typenames 0 : { *(.debug_typenames) }
.debug_varnames  0 : { *(.debug_varnames) }
/DISCARD/ : { *(.note.GNU-stack) }
```

5.9 RAM 函数

函数可以定位到 RAM 中，以提高性能。 `__ramfunc__` 和 `__longramfunc__` 说明符在函数声明中用于指定函数将在 RAM 之外执行。

启动代码会将指定为 RAM 函数的函数复制到 RAM 中，然后，对那些函数的所有调用都将引用 RAM 单元。位于 RAM 中的函数与位于程序存储器中的函数将处于不同的 512 MB 存储器段中，所以如果要从不处于 RAM 中的函数调用任意 RAM 函数，那么对于这些 RAM 函数都应应用 `longcall` 属性。 `__longramfunc__` 说明符将应用 `longcall` 属性，并将函数放入 RAM[1] 中。

```
/* function 'foo' will be placed in RAM */
void __ramfunc__ foo (void)
{
}

/* function 'bar' will be placed in RAM and will be invoked
   using the full 32 bit address */
void __longramfunc__ bar (void)
{
}
```

1. 指定 `__longramfunc__` 在功能上等价于同时指定 `__ramfunc__` 和 `__longcall__`。

注：



附录 A 实现定义的操作

A.1 简介

本章讨论在 MPLAB C32 C 编译器中，对于 “实现定义的操作”所作的选择。

A.2 主要内容

本章讨论的内容包括：

- 概述
- 翻译
- 环境
- 标识符
- 字符
- 整型
- 浮点型
- 数组和指针
- 提示
- 结构、联合、枚举和位域
- 限定符
- 声明符
- 语句
- 预处理伪指令
- 库函数
- 架构

A.3 概述

ISO C 要求，对于标准中定义为 “实现定义的”操作，符合标准的实现必须编制关于对这些操作进行选择的文档。以下几节列出了所有这些方面：针对 MPLAB C32 C 编译器所作的选择以及 ISO/IEC 9899:1999 标准中对应的小节号。

A.4 翻译

ISO 标准： “如何标识一个诊断消息 （3.10 和 5.1.1.3）。”

实现： stderr 的所有输出都是诊断消息。

ISO 标准： “在翻译阶段 3 中，空白字符 （换行符除外）组成的每个非空序列是保留还是替换为一个空格字符 （5.1.1.2）。”

实现： 每个空白字符序列替换为一个空格字符。

A.5 环境

ISO 标准：	“在独立环境中，程序启动时调用的函数的名称和类型（5.1.2.1）。”
实现：	int main (void);
ISO 标准：	“在独立环境中，终止程序的影响 （5.1.2.1）。”
实现：	执行一个无限循环指令 （转移到自身）。
ISO 标准：	“定义 `main` 函数可采用的备选方法 （5.1.2.2.1）。”
实现：	int main (void);
ISO 标准：	“为传递给 `main` 的 `argv` 参数指向的字符串提供的值（5.1.2.2.1）。”
实现：	不传递任何参数给 `main`。对 `argc` 或 `argv` 的引用未定义。
ISO 标准：	“交互式设备的要素 （5.1.2.3）。”
实现：	由应用程序定义。
ISO 标准：	“在程序启动时执行的 `signal(sig, SIG_IGN);` 的等价语句所针对的信号 （7.14.1.1）。”
实现：	信号由应用程序定义。
ISO 标准：	“当发出 `SIGABRT` 信号但未被捕获时，返回给主机环境以指示终止未成功的状态的形式 （7.20.4.1）。”
实现：	主机环境由应用程序定义。
ISO 标准：	“`exit` 函数返回给主机环境，报告终止成功或不成功的状态的形式 （7.20.4.3）。”
实现：	主机环境由应用程序定义。
ISO 标准：	“`exit` 函数的参数的值不为 `0`、 `EXIT_SUCCESS` 或 `EXIT_FAILURE` 时，它返回给主机环境的状态 （7.20.4.3）。”
实现：	主机环境由应用程序定义。
ISO 标准：	“环境名称集，以及用于更改由 `getenv` 函数使用的环境列表的方法 （7.20.4.4）。”
实现：	主机环境由应用程序定义。

ISO 标准： “系统函数执行字符串的方式 （7.20.4.5）。”

实现： 主机环境由应用程序定义。

A.6 标识符

ISO 标准： “可能出现在标识符中的附加多字节字符以及它们与通用字符名称的对应关系 （6.4.2）。”

实现： 无。

ISO 标准： “标识符中有效初始字符的数量 （5.2.4.1, 6.4.2）。”

实现： 所有字符都有效。

A.7 字符

ISO 标准： “字节中的位数 （C90 3.4 和 C99 3.6）。”

实现： 8。

ISO 标准： “执行字符集的成员的值 （C90 和 C99 5.2.1）。”

ISO 标准： “为每个标准字母表转义序列生成的执行字符集的成员的唯一值 （C90 和 C99 5.2.2）。”

实现： 执行字符集是 ASCII。

ISO 标准： “存储了除基本执行字符集成员之外的任意其他字符的 char 对象的值 （C90 6.1.2.5, C99 6.2.5）。”

实现： char 对象的值是字符在源字符集中的 8 位二进制表示。即，不进行任何转换。

ISO 标准： “signed char 和 unsigned char 中，哪一个与 “普通” char 具有相同的范围、表示和行为（C90 6.1.2.5、C90 6.2.1.1、C99 6.2.5 和 C99 6.3.1.1）。”

实现： 默认情况下， signed char 在功能上等价于普通 char。可以使用选项 `-funsigned-char` 和 `-fsigned-char` 来更改默认设置。

ISO 标准： “源字符集的成员 （字符常量和字符串文字量）到执行字符集的成员的映射 （C90 6.1.3.4、 C99 6.4.4.4、 C90 和 C99 5.1.1.2）。”

实现： 源字符集的二进制表示保存到执行字符集。

ISO 标准： “包含多个字符，或包含未映射到单字节执行字符的字符或转义序列的整型字符常量的值 （C90 6.1.3.4 和 C99 6.4.4.4）。”

实现： 编译器以每次一个字符的方式确定多字符的字符常量的值。前一个值被左移 8 位，然后下一个字符的位组合以掩码方式移入。最终的结果为 int 类型。如果结果大于 int 的表示范围，那么会发出一个警告诊断消息，并且值被截断为 int 的长度。

ISO 标准： “包含多个多字节字符，或包含不以扩展执行字符集表示的多字节字符或转义序列的宽字符常量的值 （C90 6.1.3.4 和 C99 6.4.4.4）。”

实现： 参见上文。

ISO 标准： “用于将包含单个多字节字符 （该字符映射到扩展执行字符集的一个成员）的宽字符常量转换为相应宽的字符代码的当前语言环境 （C90 6.1.3.4 和 C99 6.4.4.4）。”

实现： LC_ALL

ISO 标准： “用于将宽字符串文字量转换为相应宽的字符代码的当前语言环境 （C90 6.1.4 和 C99 6.4.5）。”

实现： LC_ALL

ISO 标准： “包含不以执行字符集表示的多字节字符或转义序列的字符串文字量的值 （C90 6.1.4 和 C99 6.4.5）。”

实现： 从源字符集保留字符的二进制表示。

A.8 整型

ISO 标准： “实现中存在的所有扩展整型 （C99 6.2.5）。”

实现： 不存在任何扩展整型。

ISO 标准： “带符号整型是使用符号与量值、 2 的补码还是 1 的补码表示，以及异常值是陷阱表示还是普通值 （C99 6.2.6.2）。”

实现： 所有整型都以 2 的补码表示，所有位组合都是普通值。

ISO 标准： “任意扩展整型相对于另一个具有相同精度的扩展整型的级别 （C99 6.3.1.1）。”

实现： 不支持任何扩展整型。

ISO 标准： “将某个整型转换为带符号整型，但前者的值无法由属于后者类型的目标表示时，所产生的结果或所发出的信号 （C90 6.2.1.2 和 C99 6.3.1.3）。”

实现： 在将值 X 转换为宽度为 N 的类型时，结果的值是 X 的 2 的补码的低 N 位。即， X 被截断为 N 位。不发出任何信号。

ISO 标准： “对带符号整型的一些位操作的结果 （C90 6.3 和 C99 6.5）。”

实现： 对带符号值进行位操作时，会对该值的 2 的补码表示 （包括符号位）进行操作。带符号值右移表达式的结果为符号扩展。

C99 允许不定义带符号数 “<<”操作的一些方面。 MPLAB C32 C 编译器不这么做。

A.9 浮点型

ISO 标准： “浮点运算的精度，以及 <math.h> 和 <complex.h> 中返回浮点型结果的库函数的精度 （C90 和 C99 5.2.4.2.2）。”

实现： 精度未知。

ISO 标准： “<stdio.h>、 <stdlib.h> 和 <wchar.h> 中的库函数执行的浮点型内部表示和字符串表示之间的转换的精度 （C90 和 C99 5.2.4.2.2）。”

实现： 精度未知。

ISO 标准： “以 FLT_ROUNDS 的非标准值为特征的舍入操作 （C90 和 C99 5.2.4.2.2）。”

实现： 不使用此类值。

ISO 标准： “以 FLT_EVAL_METHOD 的非标准负值为特征的求值方法（C90 和 C99 5.2.4.2.2）。”

实现： 不使用此类值。

ISO 标准： “当整数转换为无法精确表示原始值的浮点数时的舍入方向（C90 6.2.1.3 和 C99 6.3.1.4）。”

实现： 遵循 C99 Annex F。

ISO 标准： “当浮点数转换为较窄的浮点数时的舍入方向 （C90 6.2.1.4 和 6.3.1.5）。”

实现： 遵循 C99 Annex F。

ISO 标准： “如何为某些浮点常量选择最接近的可表示值，或者与最接近的可表示值邻近的较大或较小可表示值 （C90 6.1.3.1 和 C99 6.4.4.2）。”

实现： 遵循 C99 Annex F。

ISO 标准： “在 FP_CONTRACT pragma 未禁止浮点表达式时，是否可以以及如何使用浮点表达式 （C99 6.5）。”

实现： 该 pragma 未实现。

ISO 标准： “FENV_ACCESS pragma 的默认状态 （C99 7.6.1）。”

实现： 该 pragma 未实现。

ISO 标准： “其他的浮点异常、舍入模式、环境和分类，以及它们的宏名称 （C99 7.6, 7.12）。”

实现： 均不支持。

ISO 标准： “FP_CONTRACT pragma 的默认状态 （C99 7.12.2）。”

实现： 该 pragma 未实现。

ISO 标准： “当实际舍入结果与符合 IEC 60559 要求的实现中的数学结果相等时，是否抛出 “不精确”浮点异常 （C99 F.9）。”

实现： 未知。

ISO 标准： “当结果很小，但对于符合 IEC 60559 要求的实现足够精确时，是否抛出 “下溢”（及 “不精确”）浮点异常 （C99 F.9）。”

实现： 未知。

A.10 数组和指针

ISO 标准： “将指针转换为整数 （或反之）的结果 （C90 6.3.4 和 C99 6.3.2.3）。”

实现： 将整数强制转换为指针 （或反之）的结果使用源类型的二进制表示，并按适合于目标类型的方式重新解释。

如果源类型的长度大于目标类型，则丢弃高位。当从指针强制转换为整数时，如果源类型的长度小于目标类型，那么将对结果进行符号扩展。当从整数强制转换为指针时，如果源类型的长度小于目标类型，那么结果将根据源类型是否带符号来进行扩展。

ISO 标准： “将两个指向同一数组的两个元素的指针进行相减的结果的长度（C90 6.3.6 和 C99 6.5.6）。”

实现： 32 位带符号整数。

A.11 提示

ISO 标准： “使用寄存器存储类说明符的建议的有效程度 （C90 6.5.1, C99 6.7.1）。”

实现： 寄存器存储类说明符通常没有什么作用。

ISO 标准： “使用内联函数说明符的建议的有效程度 （C99 6.7.4）。”

实现： 如果指定了 `-fno-inline` 或 `-O0`，那么将不内联任何函数，即使函数指定了 `inline` 说明符。否则，函数可能 （也可能不）进行内联，这取决于编译器的优化推断。

A.12 结构、联合、枚举和位域

ISO 标准： “联合对象的成员使用不同类型的成员访问 （C90 6.3.2.3）。”

实现： 联合对象的相应字节解释为具有被访问成员类型的对象，而不考虑对齐或其他可能的无效条件。

ISO 标准： ““普通” int 位域是作为 signed int 位域还是作为 unsigned int 位域处理（C90 6.5.2、C90 6.5.2.1、C99 6.7.2 和 C99 6.7.2.1）。”

实现： 默认情况下，普通 `int` 位域作为带符号整数处理。通过使用 `-funsigned-bitfields` 命令行选项可以改变这种行为。

ISO 标准： “除 _Bool、 signed int 和 unsigned int 之外允许的位域类型（C99 6.7.2.1）。”

实现： 不支持任何其他类型。

ISO 标准： “位域是否可以跨越存储单元边界 （C90 6.5.2.1 和 C99 6.7.2.1）。”

实现： 不能。

ISO 标准： “位域在单元内的分配顺序 （C90 6.5.2.1 和 C99 6.7.2.1）。”

实现： 位域从左向右分配。

ISO 标准： “结构的非位域成员的对齐 （C90 6.5.2.1 和 C99 6.7.2.1）。”

实现： 每个成员根据成员类型的对齐限制，定位到所允许距离当前位置偏移量最小的位置。

ISO 标准： “与每种枚举类型兼容的整型 （C90 6.5.2.2 和 C99 6.7.2.2）。”

实现： 如果枚举值非负，那么类型为 `unsigned int`，否则为 `int`。`-fshort-enums` 命令行选项可以更改该设置。

A.13 限定符

ISO 标准： “访问具有 volatile 限定类型的对象的要素 （C90 6.5.3 和 C99 6.7.3）。”

实现： 使用 volatile 对象的值或向 volatile 对象存储值的任意表达式均视为对该对象进行访问。不能保证此类访问是原子形式的。

如果一个表达式中含有对 volatile 对象的引用，但既不使用也不存储对象的值，那么表达式是否视为对 volatile 对象的访问取决于对象的类型。如果对象属于标量类型、带有单个标量类型成员的聚合类型，或者含有 （仅）标量类型成员的联合，那么表达式视为对 volatile 对象的访问。否则，将对表达式求值，但不将其视为对 volatile 对象的访问。

例如，

```
volatile int a;

a; /* access to ‘a’ since ‘a’ is scalar */
```

A.14 声明符

ISO 标准： “可以修改算术、结构或联合类型的声明符的最大数量 （C90 6.5.4）。”

实现： 无限制。

A.15 语句

ISO 标准： “switch 语句中的 case 值的最大数量 （C90 6.6.4.2）。”

实现： 无限制。

A.16 预处理伪指令

ISO 标准： “如何将两种头文件名称形式的序列映射到头文件或外部源文件名称 （C90 6.1.7 和 C99 6.4.7）。”

实现： 定界符之间的字符序列视为一个字符串，作为主机环境的文件名。

ISO 标准： “控制条件包含的常量表达式中的字符常量的值是否与执行字符集中的相同字符常量的值匹配 （C90 6.8.1 和 C99 6.10.1）。”

实现： 是。

ISO 标准： “控制条件包含的常量表达式中的单字符常量的值是否可以具有负值 （C90 6.8.1 和 C99 6.10.1）。”

实现： 是。

ISO 标准： “所包含的以 < > 定界的头文件的搜索位置，以及如何指定这些位置或如何标识头文件 （C90 6.8.2 和 C99 6.10.2）。”

实现： `<install directory>/lib/gcc/pic32mx/3.4.4/include`

`<install directory>/pic32mx/include`

ISO 标准： “如何在所包含的以 “” 定界的头文件的搜索目录中搜索指定的源文件 （C90 6.8.2 和 C99 6.10.2）。”

实现： 编译器首先在含有包含文件的目录中、由 `-iquote` 命令行指定的目录 （如果有）中搜索指定文件，然后在以 < > 定界的头文件的搜索目录中进行搜索。

ISO 标准： “将预处理标记合并到头文件名称中的方法 （C90 6.8.2 和 C99 6.10.2）。”

实现： 所有标记 （包括空格）都视为头文件名称的一部分。对于定界符内部的标记，将不执行宏展开。

ISO 标准： “`#include` 处理的嵌套限制 （C90 6.8.2 和 C99 6.10.2）。”

实现： 无限制。

ISO 标准： “对于每个所识别的非 `STDC` `#pragma` 伪指令的操作 （C90 6.8.6 和 C99 6.10.6）。”

实现： 请参见**第 1.7 节 “属性和 Pragma 伪指令”**。

ISO 标准： “当转换的日期和时间分别不可用时， `__DATE_ _` 和 `__TIME_ _` 的定义 （C90 6.8.8 和 C99 6.10.8）。”

实现： 转换的日期和时间始终可用。

A.17 库函数

ISO 标准： “宏 NULL 展开为空指针常量 （C90 7.1.6 和 C99 7.17）。”

实现： `(void *)0`

ISO 标准： “独立程序可用的任何库工具，子句 4 所要求的最小工具集除外 （5.1.2.1）。”

实现： 请参见 *“MPLAB C32 C Compiler Libraries”* （DS51685A）。

ISO 标准： “`assert` 宏打印的诊断信息的格式 （7.2.1.1）。”

实现： “失败的断言：‘*消息*’，行 *line*，‘*文件名*’。 \n”

ISO 标准： “FENV_ACCESS pragma 的默认状态 （7.6.1）。”

实现： 未实现。

ISO 标准： “由 `fegetexceptflag` 函数存储的浮点异常标志的表示 （7.6.2.2）。”

实现： 未实现。

ISO 标准： “除了上溢或下溢异常之外， `feraiseexcept` 函数是否抛出 “不精确”异常 （7.6.2.3）。”

实现： 未实现。

ISO 标准： “除 `FE_DFL_ENV` 之外，可以用作 `fesetenv` 或 `feupdateenv` 函数参数的浮点环境宏 （7.6.4.3 和 7.6.4.4）。”

实现： 未实现。

ISO 标准： “除 “`C`” 和 “” 之外，可以作为第二个参数传递给 `setlocale` 函数的字符串 （7.11.1.1）。”

实现： 无。

ISO 标准： “当 `FLT_EVAL_METHOD` 宏的值小于 0 或大于 2 时，为 `float_t` 和 `double_t` 定义的类型 （7.12）。”

实现： 未实现。

ISO 标准： “`INFINITY` 宏展开到的最大极限 （如果有）（7.12）。”

实现： 未实现。

ISO 标准： “NAN 宏展开到的 quiet NaN （如果定义）（7.12）。”

实现： 未实现。

ISO 标准： “除了本国际标准所要求的之外，数学函数的域错误（7.12.1）。”

实现： 无。

ISO 标准： “发生域错误时数学函数返回的值，以及是否将 errno 设置为宏 EDOM 的值 （7.12.1）。”

实现： 发生域错误时， errno 设置为 EDOM。

ISO 标准： “发生上溢和 / 或下溢范围错误时，数学函数是否将 errno 设置为宏 ERANGE 的值 （7.12.1）。”

实现： 是。

ISO 标准： “FP_CONTRACT pragma 的默认状态 （7.12.2）。”

实现： 未实现。

ISO 标准： “当 fmod 的第二个参数为 0 时，是产生域错误还是返回 0（7.12.10.1）。”

实现： 返回 NaN。

ISO 标准： “在减小商时， remquo 函数使用的以 2 为基的模数对数（7.12.10.3）。”

实现： 未实现。

ISO 标准： “信号的集合，它们的语意，以及它们的默认处理 （7.14）。”

实现： 信号的默认处理是始终返回失败。实际的信号处理由应用程序定义。

ISO 标准： “在调用信号处理程序之前未执行 signal(sig, SIG_DFL); 的等价函数的情况下，所执行的信号阻塞 （7.14.1.1）。”

实现： 由应用程序定义。

ISO 标准： “在调用信号 SIGILL 的信号处理程序之前，是否执行 signal(sig, SIG_DFL); 的等价函数 （7.14.1.1）。”

实现： 由应用程序定义。

ISO 标准： “除 SIGFPE、 SIGILL 和 SIGSEGV 之外，对应于计算异常的信号值 （7.14.1.1）。”

实现： 由应用程序定义。

ISO 标准： “文本流的最后一行是否需要一个终止换行字符 （7.19.2）。”

实现： 是。

ISO 标准： “写到文本流中的紧接在换行符前面的空格字符在读入时是否出现（7.19.2）。”

实现： 是。

ISO 标准： “可附加至写入二进制流的数据的空字符数 （7.19.2）。”

实现： 不向二进制流附加任何空字符。

ISO 标准： “附加模式流的文件位置说明符最初位于文件开始还是末尾（7.19.3）。”

实现： 由应用程序定义。系统级函数 open 使用 O_APPEND 标志进行调用。

ISO 标准： “对文本流的写操作是否导致关联的文件在该点之后被截断（7.19.3）。”

实现： 由应用程序定义。

ISO 标准： “文件缓冲的特征 （7.19.3）。”

ISO 标准： “零长度文件是否确实存在 （7.19.3）。”

实现： 由应用程序定义。

ISO 标准： “构造有效文件名的规则 （7.19.3）。”

实现： 由应用程序定义。

ISO 标准： “同一文件是否可以同时打开多次 （7.19.3）。”

实现： 由应用程序定义。

ISO 标准： “文件中多字节字符所用编码的性质和选项 （7.19.3）。”

实现： 对于每个文件，编码是相同的。

ISO 标准： “remove 函数对于已打开文件的作用 （7.19.4.1）。”

实现： 由应用程序定义。系统函数 unlink 会被调用。

ISO 标准： “如果在调用 rename 函数之前，已存在具有新名称的文件，会有什么影响 （7.19.4.2）。”

实现： 由应用程序定义。系统函数 link 将会被调用来创建新文件名，然后 unlink 被调用来删除旧文件名。通常，如果新文件名已存在， link 将失败。

ISO 标准： “在程序异常终止时，是否删除打开的临时文件 （7.19.4.3）。”

实现： 否。

ISO 标准： “当 tmpnam 函数的调用次数超过 TMP_MAX 次时，会发生什么情况 （7.19.4.4）。”

实现： 将重新使用第一个临时名称。

ISO 标准： “允许哪些模式更改 （如果有），以及在哪些情况下允许更改（7.19.5.4）。”

实现： 文件通过系统级 close 函数关闭，然后使用 open 函数以新模式重新打开。除了应用程序定义的那些限制之外，对于 open 和 close 函数没有任何其他限制。

ISO 标准： “用于打印无穷大或 NaN 的样式，以及在打印 NaN 时，采用 *n-char-sequence* 时，该样式的含义 （7.19.6.1 和 7.24.2.1）。”

实现： 不打印任何字符串。

NaN 打印为 “NaN”。

无穷大打印为 “[-/+]Inf”。

ISO 标准： “fprintf 或 fwprintf 函数中 %p 转换的输出 （7.19.6.1 和 7.24.2.1）。”

实现： 在功能上等价于 %x。

ISO 标准： “在 fscanf 或 fwscanf 函数中 %[转换的扫描列表中，当 - 字符既不是第一个字符也不是最后一个字符，也不是 ^ 字符作为第一个字符时的第二个字符，此时对 - 字符的解释 （7.19.6.2 和 7.24.2.1）。”

实现： 未知。

ISO 标准： “fscanf 或 fwscanf 函数中 %p 转换所匹配的序列的集合（7.19.6.2 和 7.24.2.2）。”

实现： 与 %x 所匹配的序列集合相同。

ISO 标准： “fscanf 或 fwscanf 函数中与 %p 转换对应的输入项的解释 （7.19.6.2 和 7.24.2.2）。”

实现： 如果结果是无效指针，那么行为是未定义的。

ISO 标准： “fgetposs、 fsetpos 或 ftell 函数在失败时对宏 errno 设置的值 （7.19.9.1、 7.19.9.3 和 7.19.9.4）。”

实现： 如果结果超出 LONG_MAX，那么 errno 设置为 ERANGE。

其他错误由应用程序定义，依据是其对于 lseek 函数的定义。

ISO 标准： “在由 strtod、 strtof、 strtold、 wcstod、 wcstof 或 wcstold 函数转换的字符串中， *n-char-sequence* 的含义 （7.20.1.3 和 7.24.4.1.1）。

实现： 该字符串未附加任何含义。

ISO 标准： “发生下溢时， strtod、 strtof、 strtold、 wcstod、 wcstof 或 wcstold 函数是否将 errno 设置为 ERANGE（7.20.1.3 和 7.24.4.1.1）。”

实现： 是。

ISO 标准： “当所请求的长度为 0 时， calloc、 malloc 和 realloc 函数是返回空指针还是返回指向已分配对象的指针 （7.20.3）。”

实现： 返回指向静态分配的对象的指针。

ISO 标准： “调用 abort 函数时，是否清空打开的输出流、关闭打开的流或删除临时文件 （7.20.4.1）。”

实现： 否。

ISO 标准： “abort 函数返回给主机环境的终止状态 （7.20.4.1）。”

实现： 默认情况下，不存在主机环境。

ISO 标准： “system 函数在其参数不是空指针时返回的值 （7.20.4.5）。”

实现： 由应用程序定义。

ISO 标准： “本地时区和夏令时 （7.23.1）。”

实现： 由应用程序定义。

ISO 标准： “clock 函数的年代 （7.23.2.1）。”

实现： 由应用程序定义。

ISO 标准： “在标准化的 tmx 结构中， tm_isdst 的正值 （7.23.2.6）。”

实现： 1。

ISO 标准： “在 “C” 语言环境中， strftime、 strfxtime、 wcsftime 和 wcsfxtime 函数的 %Z 说明符的替换字符串 （7.23.3.5、 7.23.3.6、 7.24.5.1 和 7.24.5.2）。”

实现： 未实现。

ISO 标准： “在符合 IEC 60559 要求的实现中，三角函数、双曲函数、以 *e* 为底的指数函数、以 *e* 为基的对数函数、错误函数和对数 gamma 函数是否抛出 “不精确”异常，以及何时抛出 （F.9）。”

实现： 否。

ISO 标准： “在符合 IEC 60559 要求的实现中，当舍入结果实际等于数学结果时，是否抛出 “不精确”异常 （F.9）。”

实现： 否。

ISO 标准： “在符合 IEC 60559 要求的实现中，当结果很小，但足够精确时，是否抛出下溢 （及 “不精确”）异常 （F.9）。”

实现： 否。

ISO 标准： “函数是否遵从舍入方向模式 （F.9）。”

实现： 并不强制要求遵从舍入模式。

A.18 架构

ISO 标准： “为在头文件 <float.h>、 <limits.h> 和 <stdint.h> 中指定的宏分配的值或表达式（C90 和 C99 5.2.4.2、 C99 7.18.2，以及 C99 7.18.3）。

实现： 请参见**第 1.5.6 节 “limits.h”**。

ISO 标准： “任意对象中的字节的数量、顺序和编码 （当未在标准中明确指定时）（C99 6.2.6.1）。”

实现： 小尾数形式，从最低字节开始填充。请参见**第 1.5 节 “数据存储”**。

ISO 标准： “sizeof 操作符的结果 （C90 6.3.3.4 和 C99 6.5.3.4）。”

实现： 请参见**第 1.5 节 “数据存储”**。

注：

附录 B 开源许可

B.1 简介

本章简要介绍用于 MPLAB C32 C 编译器软件包的一些部分的开源许可证。

B.2 通用公共许可证

编译器、汇编器、链接器和关联的二进制实用程序的可执行文件受 GNU 通用公共许可证保护。请参见产品安装目录中的文件 doc/COPYING.GPL，获取许可证的完整文本。

B.3 BSD 许可证

标准函数库的一些部分按加利福尼亚大学的 “BSD”许可证的条款分发：

Copyright © Regents of the University of California.

版权所有。

在满足下列条件的前提下，允许再发布和使用经过或未经过修改的、源代码或二进制形式的本软件：

1. 源代码的再发布，必须保留上面的版权声明、这几条许可条件，以及下面的免责声明。

2. 二进制形式的再发布，必须在随同提供的文档和 / 或其他材料中，复制上面的版权声明、这几条许可条件，以及下面的免责声明。

3. 一切提及本软件特性或使用的广告材料必须显示以下内容：

本产品包含由加利福尼亚大学伯克利分校及其贡献者开发的软件。

4. 未经事先书面特别许可，加利福尼亚大学的名称及其贡献者名字，均不得用于担保或宣传从本软件派生的产品。

本软件由版权所有者和贡献者 “按现状”提供。不提供明示或暗示的担保，包括但不限于关于适销性和特定用途的适用性的暗示担保。在任何情况下，由于使用本软件造成的直接、间接、连带、特别、惩戒或因此而造成的损害 （包括但不限于获得替代品及服务、无法使用、丢失数据、损失盈利或业务中断），无论此类损害是如何造成的，基于何种责任推断，是否属于合同范畴，严格赔偿责任或民事侵权行为 （包括疏忽和其他原因），即使预先被告知此类损害可能发生，版权所有者和贡献者均不承担任何责任。

B.4 SUN MICROSYSTEMS

标准函数库的一些部分版权归 Sun Microsystems 所有，按以下条款授予的许可分发：

开发于 SunPro （Sun Microsystems, Inc. 子公司）。在保留本声明的前提下，允许自由使用、复制、修改和分发本软件。



索引

Z

注：



全球销售及服务网点

美洲

公司总部 Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 1-480-792-7200
Fax: 1-480-792-7277
技术支持：
http://support.microchip.com
网址：www.microchip.com

亚特兰大 Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

波士顿 Boston
Westborough, MA
Tel: 1-774-760-0087
Fax: 1-774-760-0088

芝加哥 Chicago
Itasca, IL
Tel: 1-630-285-0071
Fax: 1-630-285-0075

达拉斯 Dallas
Addison, TX
Tel: 1-972-818-7423
Fax: 1-972-818-2924

底特律 Detroit
Farmington Hills, MI
Tel: 1-248-538-2250
Fax: 1-248-538-2260

科科莫 Kokomo
Kokomo, IN
Tel: 1-765-864-8360
Fax: 1-765-864-8387

洛杉矶 Los Angeles
Mission Viejo, CA
Tel: 1-949-462-9523
Fax: 1-949-462-9608

圣克拉拉 Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

加拿大多伦多 Toronto
Mississauga, Ontario,
Canada
Tel: 1-905-673-0699
Fax: 1-905-673-6509

亚太地区

亚太总部 Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

中国 - 北京
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

中国 - 成都
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

中国 - 香港特别行政区
Tel: 852-2401-1200
Fax: 852-2401-3431

中国 - 南京
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

中国 - 青岛
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

中国 - 上海
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

中国 - 沈阳
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

中国 - 深圳
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

中国 - 武汉
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

中国 - 厦门
Tel: 86-592-238-8138
Fax: 86-592-238-8130

中国 - 西安
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

中国 - 珠海
Tel: 86-756-321-0040
Fax: 86-756-321-0049

台湾地区 - 高雄
Tel: 886-7-536-4818
Fax: 886-7-536-4803

台湾地区 - 台北
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

台湾地区 - 新竹
Tel: 886-3-572-9526
Fax: 886-3-572-6459

亚太地区

澳大利亚 Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

印度 India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

印度 India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

印度 India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

日本 Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

韩国 Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

韩国 Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 或
82-2-558-5934

马来西亚 Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

马来西亚 Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

菲律宾 Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

新加坡 Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

泰国 Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

欧洲

奥地利 Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

丹麦 Denmark-Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

法国 France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

德国 Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

意大利 Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

荷兰 Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

西班牙 Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

英国 UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/02/08